USENIX Association

Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’16)

March 16–18, 2016
Santa Clara, CA, USA

Conference Organizers

Program Co-Chairs

Katerina Argyraki, *EPFL*

Rebecca Isaacs, *Google*

Program Committee

Aditya Akella, *University of Wisconsin—Madison*

Mohammad Alizadeh, *Massachusetts Institute of Technology*

Mona Attariyan, *Google*

Hari Balakrishnan, *Massachusetts Institute of Technology*

Mahesh Balakrishnan, *Yale University*

Aruna Balasubramanian, *Stony Brook University*

Sujata Banerjee, *HP Labs*

Paul Barford, *University of Wisconsin—Madison and comScore*

Ranjita Bhagwan, *Microsoft Research India*

Nathan Bronson, *Facebook*

Paolo Costa, *Microsoft Research*

Paul Francis, *Max Planck Institute for Software Systems (MPI-SWS)*

Monia (Manya) Ghobadi, *Microsoft Research*

Shyam Gollakota, *University of Washington*

Jon Howell, *Google*

Kyle Jamieson, *Princeton University*

Srikanth Kandula, *Microsoft*

Brad Karp, *University College London*

S. Keshav, *University of Waterloo*

Changhoon Kim, *Barefoot Networks*

Ramakrishna Kotla, *Amazon*

Jinyang Li, *New York University*

David Lie, *University of Toronto*

Kate C.-J. Lin, *Academia Sinica, Taiwan*

Wyatt Lloyd, *University of Southern California*

Jay Lorch, *Microsoft Research*

Ratul Mahajan, *Microsoft Research*

Prateek Mittal, *Princeton University*

Thomas Moscibroda, *Microsoft Research*

David Oran, *Cisco Systems*

Oriana Riva, *Microsoft Research*

Vyas Sekar, *Carnegie Mellon University*

Siddhartha Sen, *Microsoft Research*

Srinivasan Seshan, *Carnegie Mellon University*

Ankit Singla, *ETH Zürich*

Jonathan Smith, *University of Pennsylvania*

Alex Snoeren, *University of California, San Diego*

Kobus Van der Merwe, *University of Utah*

Laurent Vanbever, *ETH Zürich*

Matt Welsh, *Google*

Poster Session Co-Chairs

Aruna Balasubramanian, *Stony Brook University*

Laurent Vanbever, *ETH Zürich*

Steering Committee

Paul Barham, *Google*

Nick Feamster, *Georgia Institute of Technology*

Casey Henderson, *USENIX Association*

Arvind Krishnamurthy, *University of Washington*

Brian Noble, *University of Michigan*

Jennifer Rexford, *Princeton University*

Mike Schroeder

Alex C. Snoeren, *University of California, San Diego*

External Reviewers

Nishanth Chandran

Yu-han (Tiffany) Chen

Mike Freedman

Brighten Godfrey

Peter Iannucci

Eddie Kohler

Ravi Netravali

Amy Ousterhout

Jonathan Perry

Anirudh Sivaraman

John Wilkes

NSDI '16: 13th USENIX Symposium on Networked Systems Design and Implementation March 16–18, 2016 Santa Clara, CA

Wednesday, March 16, 2016

Network Architectures and Protocols

Content Delivery

Thursday, March 17, 2016

Wireless I

Flexible Networks

Dependability and Monitoring

Resource Sharing

Friday, March 18, 2016

Distributed Systems

In-Network Processing

(Friday March 18, continues on next page)

Security and Privacy

Wireless II

Message from the
NSDI ’16 Program Co-Chairs

A warm welcome to NSDI ‘16! We are delighted to continue the NSDI tradition and share with you the latest and greatest research on network systems. This year’s program features new ways to deliver content, respect user privacy, improve network dependability and flexibility, and share network resources, as well as some very exciting new wireless technologies. Moreover, this year’s Operational Systems Track—which describes experience with real, deployed networks—features work on distributed stream processing, software load balancing, cellular traffic optimization, and social networks.

We received 225 submissions and accepted 45 papers. Our Program Committee consisted of 42 members with a mix of research and industry experience. We completed two rounds of reviews: in the first round, each paper received 3 reviews; the 101 papers that advanced to the second round received at least two more reviews. Once the reviewing process was over, the committee engaged in online discussion and selected 72 papers that were discussed further at the PC meeting.

It has been a great pleasure working with many other people to put this program together. We would firstly, and most importantly, like to thank the authors of all submitted papers for choosing to send work of such high calibre to NSDI. Thanks also to the Program Committee for their professionalism, diligence and enthusiasm, and special thanks to Aruna Balasubramanian and Laurent Vanbever for serving as poster chairs, as well as to Srikanth Kandula and Laurent (again), for graciously agreeing to a last-minute request to take on extra reviewing load. We are also very grateful to the USENIX staff, especially Casey and Michele, whose helpfulness knows no bounds. Finally, NSDI wouldn’t be what it is without the attendees, so thank you very much for being here. We hope you enjoy the conference!

Katerina Argyraki, *EPFL*
Rebecca Isaacs, *Google*
NSDI ’16 Program Co-Chairs

An Industrial-Scale Software Defined Internet Exchange Point

Arpit Gupta[⋆], Robert MacDavid[⋆], Rüdiger Birkner[†],
Marco Canini[⋄], Nick Feamster[⋆], Jennifer Rexford[⋆], Laurent Vanbever[†]
[⋆]Princeton University [†]ETH Zürich [⋄]Université catholique de Louvain
http://sdx.cs.princeton.edu/

Abstract

Software-Defined Internet Exchange Points (SDXes) promise to significantly increase the flexibility and function of interdomain traffic delivery on the Internet. Unfortunately, current SDX designs cannot yet achieve the scale required for large Internet exchange points (IXPs), which can host hundreds of participants exchanging traffic for hundreds of thousands of prefixes. Existing platforms are indeed too slow and inefficient to operate at this scale, typically requiring minutes to compile policies and millions of forwarding rules in the data plane.

We motivate, design, and implement iSDX, the first SDX architecture that can operate at the scale of the largest IXPs. We show that iSDX reduces both policy compilation time and forwarding table size by two orders of magnitude compared to current state-of-the-art SDX controllers. Our evaluation against a trace from one of the largest IXPs in the world found that iSDX can compile a realistic set of policies for 500 IXP participants in less than three seconds. Our public release of iSDX, complete with tutorials and documentation, is already spurring early adoption in operational networks.

1 Introduction

Software-Defined Networking (SDN) has reshaped the design of many networks and is poised to enable new capabilities in interdomain traffic delivery. A natural place for this evolution to occur is at Internet exchange points (IXPs), which are becoming increasingly prevalent, particularly in developing regions. Because many Autonomous Systems (ASes) interconnect at IXPs, introducing flexible control at these locations makes it easier for them to control how traffic is exchanged in a direct, and more fine-grained way. In previous work [14], we offered an initial design of a Software-Defined Internet Exchange Point (SDX) and showed how introducing SDN functionality at even a single IXP can catalyze new traffic-management capabilities, ranging from better inbound traffic engineering to application-specific peering and server load balancing.

Since we introduced SDX [13], many organizations and networks have built different versions of this concept [4, 14, 22, 23, 35]. Yet, many of these deployments remain relatively small-scale or limited in scope because current switch hardware cannot support large forwarding tables, and because efficiently combining the policies of independently operated networks as routes and policies change presents a significant scaling challenge.

In this paper, we tackle these scalability challenges with the design and implementation of iSDX, an industrial-scale SDX that can support interconnection for the largest IXPs on the Internet today. We design mechanisms that allow the number of participants, BGP routes, and SDN policies to scale, even for the limited table sizes of today's switches. We develop algorithms for compiling traffic control policies at the scale and speed that networks that such an IXP would require. We have implemented these algorithms in Ryu [31], a widely used SDN controller. We have released our implementation to the public with documentation and tutorials; one large government agency has tested iSDX with hardware switches and is using our controller as the basis for a deployment.

In the design and implementation of iSDX, we address two scalability challenges that are fundamental to *any* SDX design. The first challenge relates to how the control plane combines the policies of individual networks into forwarding entries in the data plane. Compiling traffic control policies expressed in a higher-level policy language to forwarding table entries can be slow, since this process involves composing the policies of multiple participants into a single coherent set of forwarding-table entries. This slow process is exacerbated by the fact that any change to BGP routing may change forwarding behavior; existing SDX designs trigger recompilation every time a BGP best route changes, which is not tractable in practice. The main scalability challenge thus involves efficiently composing the policies of individual participants, and ensuring that the need to recompile the forwarding table entries is completely decoupled from (frequent) BGP routing changes.

To scale the control plane, we introduce a new design that exploits the fact that each participant expresses its SDN policy independently, which implies that each participant can also compile its SDN policies independently, as well. This change enables more aggressive compression of the forwarding tables than is possible when all of the policies are compressed together and also allows for participant policies to be compiled in parallel. As a result, iSDX compiles the forwarding tables two orders of magnitude faster than the existing approaches; the tables are also two orders of magnitude smaller, making them suitable for practical hardware-switch deployments.

The second challenge relates to the data plane: the number of forwarding table entries that might go into the forwarding table at an IXP switch can quickly grow unacceptably large. Part of the challenge results from the fact that the policies that each network writes have to be consistent with the BGP routes that each participant advertises, to ensure that an SDN policy cannot cause the switch to forward traffic on a path that was never advertised in BGP. This process significantly inflates the number of forwarding table entries in the switch and is a considerable deployment hurdle. Large industrial-scale IXPs can have over 700 participants exchanging traffic for hundreds of thousands of prefixes; combined with the fact that each of these participants may now introduce policies for specific traffic flows, the number of forwarding table entries quickly becomes intractable. Although our initial design [14] reduced the size of the forwarding tables, we show that the size of these tables remained prohibitively large for industrial-scale deployments.

To address the data-plane challenge, we introduce an efficient encoding mechanism where the IXP fabric forwards the packet based on an opaque tag that resides in the packet's destination MAC field. This tag explicitly encodes both the next-hop for the packet and the set of ASes that advertise BGP routes for the packet's destination, thus making it possible to remove this information from the switch tables entirely. This separation prevents BGP routing updates from triggering recomputation and recompilation of the forwarding table entries. Using features in OpenFlow 1.3 that support matching on fields with arbitrary bitmasks, we significantly reduce the size of this table by grouping tags with common bitmasks.

In summary, we present the following contributions:

- The design and implementation of iSDX, the first SDX controller that scales to large industrial-scale IXPs. We devised new mechanisms for distributing control-plane computation, compressing the forwarding tables, and responding to BGP routing changes, reducing the compilation time and forwarding table size by several orders of magnitude. (Sections 3–5)
- A public, open-source implementation of iSDX on Github [16]; the system is based on Ryu, a widely used SDN controller, and is accompanied with tutorials and instructions that have already helped spur early adoption. (Section 6)
- An extensive evaluation of iSDX's scalability characteristics using a trace-driven evaluation from one of the largest IXPs in the world. Our evaluation both demonstrates that iSDX can scale to the largest IXPs and provides insight into specifically how (and to what extent) each of our optimizations and algorithms helps iSDX scale. (Section 7)

We survey related work in Section 8 and conclude in Section 9 with a discussion of open issues in SDX design and avenues for future work.

2 SDX: Background & Scaling Challenges

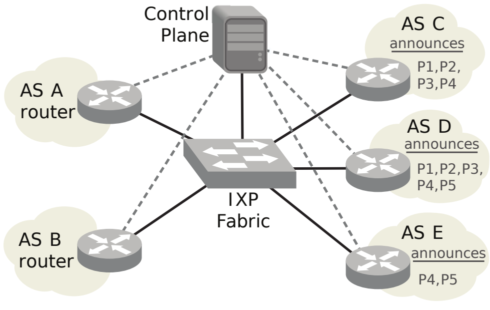
We begin with a background on our previous SDX designs [14, 35] and a demonstration that these designs cannot scale to industrial IXPs.

2.1 Background

Brief overview of SDX. An SDX is an IXP consisting of a programmable SDN fabric, coupled with a BGP route server (which allows IXP participants to exchange reachability information via BGP) and an SDN controller (which allows participants to override default BGP routing behavior with more fine-grained SDN policies). The SDX controller provides each participant AS with the abstraction of a dedicated switch that it can program using match-action policies to control traffic flows. Participants may express SDN policies on both their inbound and outbound traffic; the SDX controller ensures that no SDN policy results in traffic being forwarded to a neighboring AS that did not advertise a BGP route for the prefix that matches the packet's destination IP address.

Each participant runs an SDN control application on the central controller and has its border router exchange BGP update messages with the IXP's route server. The SDN controller combines the SDN policies from all participants, reconciles the resulting policy with the BGP routing information, and computes and installs the resulting forwarding table entries in the IXP fabric. To avoid having forwarding entries for all prefixes, our original SDX design relied on the participants' border routers to tag packets entering the IXP fabric with a *forwarding equivalence class* of destination prefixes with the same forwarding action. For backwards compatibility, the tag was the destination MAC address, set in response to the border router sending an ARP query for the next-hop IP address from the BGP route advertisement. The SDX route server computed a different (virtual) next-hop IP address for each equivalence class of prefixes to trigger the border router to use a common MAC address for packets sent to the group of destination IP addresses.

Example operation. Figure 1a shows an example topology with five participants; Figure 1b shows the routes advertised to *A* and *B* and the BGP routes that they select



(a) *Example Topology*

	A	B
P1	C, **D**	**C**, D
P2	C, **D**	**C**, D
P3	C, **D**	**C**, D
P4	C, **D**, E	C, D, **E**
P5	D, **E**	D, **E**

(b) *Reachability and Next Hops (in bold) for AS A and AS B*

Figure 1: *An example with five IXP participants. Two participants AS A and AS B have outbound policies. The other three advertise five IP prefixes to both these participants.*

for each prefix (in bold). Both A and B express outbound policies. To ensure that SDN policies cause the IXP to forward traffic in a way that is consistent with the advertised BGP routes, the SDX controller *augments* each outbound policy with the reachability information. Intuitively, augmentation restricts forwarding policies so that traffic is forwarded only on paths that correspond to BGP routes that the participant has learned.

For example, suppose that A has the following outbound policies:

$\texttt{dPort=443} \rightarrow \texttt{fwd}(C)$

$\texttt{dPort=22} \rightarrow \texttt{fwd}(C)$

$\texttt{dPort=80} \land \texttt{sIp=10/24} \rightarrow \texttt{fwd}(D)$

$\texttt{dPort=80} \land \texttt{sIp=40/24} \rightarrow \texttt{fwd}(D)$

These policies forward traffic based on values of packet header fields, overriding BGP behavior. For instance, the first policy specifies HTTPS traffic (dPort=443) should be forwarded to C. Without augmentation, A would also forward the HTTPS traffic destined for prefix $P5$ to C, even though C never advertised a path for $P5$ to A. In our example, A's policies are then augmented as follows:

$\texttt{dIp} \in \{\mathbf{P1},\mathbf{P2},\mathbf{P3},\mathbf{P4}\} \land \texttt{dPort=443} \rightarrow \texttt{fwd}(C)$

$\texttt{dIp} \in \{\mathbf{P1},\mathbf{P2},\mathbf{P3},\mathbf{P4}\} \land \texttt{dPort=22} \rightarrow \texttt{fwd}(C)$

$\texttt{dIp} \in \{\mathbf{P1},\mathbf{P2},\mathbf{P3},\mathbf{P4},\mathbf{P5}\} \land \texttt{dPort=80} \land \texttt{sIp=10/24} \rightarrow \texttt{fwd}(D)$

$\texttt{dIp} \in \{\mathbf{P1},\mathbf{P2},\mathbf{P3},\mathbf{P4},\mathbf{P5}\} \land \texttt{dPort=80} \land \texttt{sIp=40/24} \rightarrow \texttt{fwd}(D)$

Augmentation enforces that the destination IP (dIp) matches one of the prefixes that either C or D announces to A, therefore ensuring congruence with BGP routing. Observe that a straightforward realization of this policy requires one distinct match-action rule for each of the five prefixes. Hence, the augmented policies would result in 18 forwarding rules instead of the four rules necessary to implement the original policy.

Similarly, if B's outbound policy is:

$\texttt{dPort=443} \rightarrow \texttt{fwd}(E)$

the SDX controller augments the policy, doubling the number of necessary rules, as follows:

$\texttt{dIp} \in \{\mathbf{P4},\mathbf{P5}\} \land \texttt{dPort=443} \rightarrow \texttt{fwd}(E)$

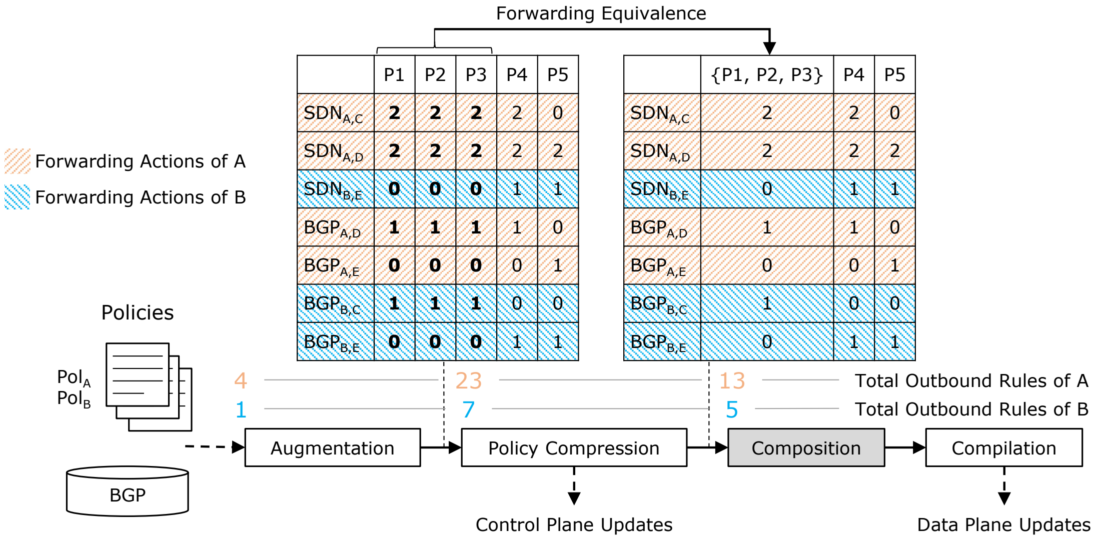
To better illustrate the scalability challenge, we capture the expansion of the switch forwarding tables using an *augmentation matrix* (Figure 2, left matrix). In this matrix, a row labeled as $\text{SDN}_{X,Y}$ refers to an SDN policy written by X that results in traffic being forwarded to Y, while columns refer to IP prefixes. The value of an element (i, j) indicates the number of forwarding table entries (*i.e.*, match-action rules) in participant i's policy where prefix j appears. Similarly, $\text{BGP}_{X,Y}$ indicates whether X selects Y as the next hop for some BGP-advertised prefix, and element (i, j) is 1 if participant A selects the route advertised by B for the prefix corresponding to column j.

For example, the element in row $\text{SDN}_{A,C}$ and column $P1$ reflects the fact there are two forwarding table entries that correspond to prefix $P1$: one for traffic with dPort=443 and one for traffic with dPort=22. The same applies for columns $P2$, $P3$, and $P4$. We can determine the total number of forwarding table entries (and the number contributed by each participant) by summing up the corresponding elements in the matrix. We will use this notation to describe compression techniques (and their effects) throughout the paper.

Previously developed compression techniques. Intuitively, the number of forwarding rules increases as the number of SDX participants with outbound policies increases (more rows) and as forwarding policies are defined for additional prefixes (more columns). To limit the number of forwarding rules, the original SDX design [14] identified the Minimum Disjoint Set (MDS) of prefixes (columns) with the same SDN policies and grouped each equivalent set into a Forwarding Equivalence Class (FEC). In the rest of this paper, we refer to this algorithm as *MDS compression*. For instance, in the preceding example, prefixes $P1, P2, P3$ belong to the same FEC, as indicated by the boldface entries in the left matrix in Figure 2. MDS compression reduces the number of forwarding table entries by assigning a virtual next-hop to each FEC, rather than to each individual prefix. Figure 2 also depicts the number of forwarding table entries before and after MDS compression. In particular, MDS compression reduces the number of columns from the total number of prefixes (5) to the number of FECs (3).

2.2 Existing SDX Designs Do Not Scale

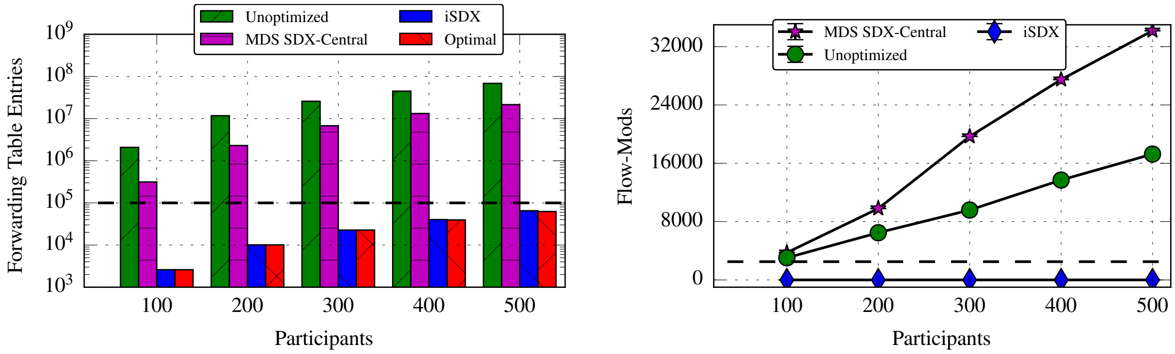
In this section, we show that existing SDX designs do not scale to the demands of industrial-scale IXPs. We explore two different state-of-the-art SDX designs: (1) an



	P1	P2	P3	P4	P5
$SDN_{A,C}$	**2**	**2**	**2**	2	0
$SDN_{A,D}$	**2**	**2**	**2**	2	2
$SDN_{B,E}$	**0**	**0**	**0**	1	1
$BGP_{A,D}$	**1**	**1**	**1**	1	0
$BGP_{A,E}$	**0**	**0**	**0**	0	1
$BGP_{B,C}$	**1**	**1**	**1**	0	0
$BGP_{B,E}$	**0**	**0**	**0**	1	1

	{P1, P2, P3}	P4	P5
$SDN_{A,C}$	2	2	0
$SDN_{A,D}$	2	2	2
$SDN_{B,E}$	0	1	1
$BGP_{A,D}$	1	1	0
$BGP_{A,E}$	0	0	1
$BGP_{B,C}$	1	0	0
$BGP_{B,E}$	0	1	1

Figure 2: *Matrix representation of AS A and AS B's outbound policies after augmentation and policy compression, as well as the stages of compression and composition in the original SDX design; the composition stage is grey to indicate that the iSDX eliminates this stage entirely.*



(a) *Number of Forwarding Table Entries.*

(b) *Data-Plane Update Rate.*

Figure 3: *Existing SDX designs can require to maintain millions of forwarding entries (left) and update 10,000s of updates per second (right). Such numbers are far from current hardware capabilities. As an illustration, the dashed line highlights the hardware capabilities of state-of-the-art SDN switches [26].*

unoptimized SDX that does not compress policies, such as that used by Google's Cardigan SDX [38]; (2) a simple, centralized SDX controller that applies MDS compression, as in our previous work [14]. We also preview the results from this paper, showing that our new architecture, iSDX, reduces the compilation time, number of forwarding table entries, and data-plane update rate by more than two orders of magnitude, thus making operation in an industrial-scale IXP practical. In each case, we evaluate the time to compute the forwarding table entries, the number of forwarding table entries, and the rate at which changes in BGP routing information induce changes in the forwarding table entries. We use a real BGP trace from one the largest IXPs in the world for this evaluation. Section 7 provides details about our experiment setup.

	Unoptimized	**Centralized MDS-SDX [14]**	**iSDX**
Time (s)	4572.15	1740.93	2.82

Table 1: *Median time (for 60 trials) to compute forwarding table entries for an IXP with 500 participants. The iSDX column shows the results for this paper.*

Existing SDX designs can take minutes to compute forwarding table entries. Table 1 shows the median time over 60 trials to compute forwarding table entries for an IXP with 500 participants for two state-of-the-art SDX designs, as well as for iSDX, the design that we present in this paper. iSDX reduces the average time to compute forwarding table entries from 30 minutes to *less than three seconds.*

Existing SDX designs can require millions of forwarding table entries. Figure 3a shows how the number of forwarding table entries increases as the number of participants increases from 100 to 500. MDS compression reduces the number of entries by an order of magnitude, but the forwarding table is still too large for even the most high-end hardware switches, which have about 100,000 TCAM entries [26]. The iSDX design ensures that the number of forwarding table entries is approximately the number of SDN policies that each participant expresses (shown as "optimal" in Figure 3a), thus allowing the number of forwarding table entries to be in the tens of thousands, rather than tens of millions.

Existing SDX designs require hundreds of thousands updates per second to the data plane. Figure 3b shows the worst-case data-plane update rate that an SDX controller must sustain to remain consistent with existing BGP updates. The update rates of existing designs are several orders of magnitude above what even top-of-the-line hardware switches can support [26] (*i.e.*, about 2,500 updates per second). In constrast, iSDX usually eliminates forwarding table updates in response to BGP updates.

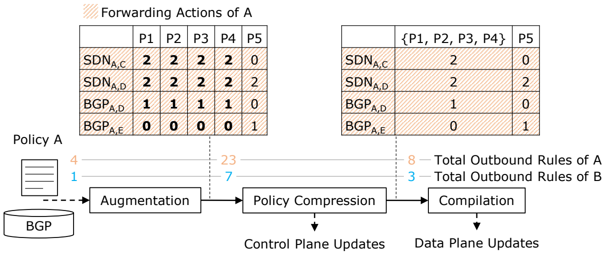
3 Design of an Industrial-Scale SDX

We introduce the design of an industrial-scale SDX (iSDX), which relies on two principles to reduce compilation time, the number of forwarding table entries, and forwarding table update rates.

3.1 Partition Control-Plane Computation

Problem: Considering all policies together reduces opportunities for compression. Centralized SDX controllers perform control-plane computations for all IXP participants. Doing so not only forces the controller to process a large single combined policy, it also creates dependencies between the policies of individual IXP participants. For example, a change to any participant's inbound policy triggers the recompilation of the policies of *all* participants who forward traffic to that participant. This process requires significant computation and also involves many (and frequent) updates to the forwarding table entries at the IXP switch.

Solution: Partition computation across participants. We solve this problem by partitioning the control-plane computation across participants. Doing so ensures that participant policies stay independent from each other. In addition, partitioning the computation enables more efficient policy compression by operating on smaller state, reducing both computation time and data plane state. Partitioning the control-plane computation among participants also enables policy compilation to scale out as the number of IXP participants and routes grows. Section 4 details this approach.



	P1	P2	P3	P4	P5
$SDN_{A,C}$	**2**	**2**	**2**	**2**	0
$SDN_{A,D}$	**2**	**2**	**2**	**2**	2
$BGP_{A,D}$	**1**	**1**	**1**	**1**	0
$BGP_{A,E}$	**0**	**0**	**0**	**0**	1

	{P1, P2, P3, P4}	P5
$SDN_{A,C}$	2	0
$SDN_{A,D}$	2	2
$BGP_{A,D}$	1	0
$BGP_{A,E}$	0	1

Figure 4: *Partitioning the Control-Plane Computation.*

3.2 Decouple BGP and SDN Forwarding

Problem: Frequent BGP updates trigger recompilation. Coupling BGP and SDN policies during compilation inflates the number of resulting forwarding table entries and also implies that any change to BGP routing triggers recompilation of the forwarding table entries, which is costly. Our previous design partially addressed this problem, but this design still requires millions of flow rules in the data plane as shown in Figure 3a. Additionally, our previous approach to reduce the number of forwarding table entries *increases* the forwarding table update rates, since any change in BGP routing may affect how entries are compressed.

Solution: Encode BGP reachability information in a separate tag. We address this problem by encoding all information about BGP best routes (and corresponding next hops) into the destination MAC addresses, which reduces the number of forwarding table entries, as well as the number of changes to the forwarding table when BGP routes change. Section 5 discusses our approach in detail.

4 Partitioning Control-Plane Computation

To achieve greater compression of the rule matrix, we need to reduce the constraints that determine which prefixes belong to the same FEC. Rather than computing one set of equivalence classes for the entire SDX, iSDX computes *separate FECs for each participant*. We first discuss how partitioning by participant reduces the size of the rule matrices and, as a side benefit, allows for faster computation. We then describe how we use multiple match-action tables and ARP relays to further improve scalability, setting the stage for further optimizations in Section 5.

4.1 Partitioning the FEC Computation

Figure 4 shows similar compression and compilation steps as the ones done in Figure 2, with the important distinction that it takes place on behalf of participant *A* only; similar operations take place on behalf of other participants. Fig-

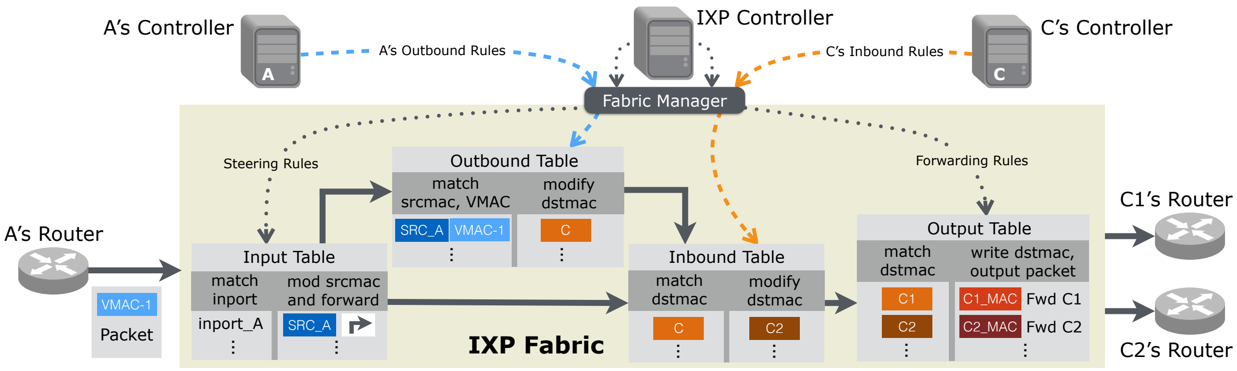


Figure 5: *Distributing forwarding rules and tags.*

ure 4 highlights two important benefits of partitioning the computation of FEC across participants:

- Computing separately for each participant reduces the number of next-hops, leading to a smaller number of larger forwarding equivalence classes. In Figure 4, the number of columns reduces from five to two.
- The computational complexity of computing FECs is proportional to the number of rows times the number of columns in the rule matrix. Now, each rule matrix is smaller, and the computation for different participants can be performed in parallel.

In practice, the SDX controller could compute the FECs for each participant, or each participant could run its own controller for computing its own FECs. In the rest of the paper, we assume each participant runs its own controller for computing its FECs.

4.2 Distributing Forwarding Rules & Tags

In addition to computing the FECs for each participant, the iSDX must realize these policies in the data plane.

Decomposing the IXP fabric into four tables: To forward traffic correctly, an SDX must combine the inbound and outbound policies for all of the participants. Representing the combination of policies in a single forwarding table, as in an OpenFlow 1.0 switch, would be extremely expensive. Some existing SDN controllers perform this type of composition [25, 33]—essentially computing a cross product of the constituent policies—and, in fact, our original SDX followed this approach [14]. Computing the cross product leads to an explosion in the number of rules, and significant recomputation whenever one of the participant policies changes.

Fortunately, modern switches have multiple stages of match-action tables, and modern IXPs consist of multiple switches. The iSDX design capitalizes on this trend. The main challenge is to determine *how* to most effectively map policies to the underlying tables.

A strawman solution would be to use a two-table pipeline, where packets first enter an outbound table implementing outbound policies for the participant where the traffic originates, followed by an inbound table that applies inbound policies for the participant that receives the traffic as it leaves the IXP fabric. Using only two tables, however, would mean that some of these tables would need to be much larger; for example, the outbound table would need to represent the cross product of all input ports and outbound policies. Additionally, using only two tables makes it more difficult to scale-out the iSDX as the number of participants grows.

As such, our design incorporates an input table, which handles all the incoming traffic and tags it with a new source MAC address based on the packet's incoming port, so that packets can be multiplexed to the outbound table. As the packet leaves the iSDX, it passes through an output table, which looks up the packet's tag in the destination MAC field and both performs the appropriate action and rewrites the packet's destination MAC address. Separate input and output tables provide a cleaner separation of function between the modules that write to each table, avoid cross-product explosion of policies, and facilitates scale-out by allowing the inbound and outbound tables to reside on multiple physical switches in the IXP infrastructure. (Such scale-out techniques are beyond of the scope of this paper.)

Figure 5 shows how the IXP fabric forwards a packet, while distributing the compilation and compression of policies across separate tables. Based on the destination IP address of the packet, suppose that AS *A*'s controller selects a route to the packet's destination via AS *D*; this route will correspond to a next-hop IP address. AS *A*'s controller will make a BGP announcement advertising this path. AS *A*'s router will issue an ARP query for the advertised next-hop IP address, and then AS *A*'s controller will respond via the ARP relay setting a virtual MAC address (in Figure 5, "VMAC-1") as the packet's destination MAC address.

When the packet enters the IXP fabric, the input table matches on the packet's incoming port and rewrites the source MAC address to indicate that the packet arrived from AS A ("SRC_A"). If A has an outbound policy, the packet will match on ("SRC_A"), and the outbound table will apply an outbound policy. If A has no outbound policy for this packet, the input table forwards the packet directly to the inbound table without changing the destination MAC. This bypass is not strictly necessary but avoids an additional lookup for packets that do not have a corresponding outbound policy. A's outbound policy thus overwrites default BGP forwarding decision and modifies the destination MAC address to "C". The inbound table rewrites the tag to correspond to the final disposition of the packet ("C1" or "C2"), which is implemented in the output table. The output table also rewrites the tag to the receiver's physical MAC address before forwarding.

Reducing ARP traffic overhead. Partitioning the FEC computation reduces the number of FECs per participant, but may increase the *total* number of FECs across all participants (*i.e.*, the number of columns across all rule matrices). To reduce the size of the forwarding tables, each data packet carries a tag (*i.e.*, a virtual MAC address) that identifies its FEC. The participant's border router learns the virtual MAC address through an ARP query on the BGP next-hop IP address of the associated routes. The use of broadcast for ARP traffic, combined with the larger number of next-hop IP addresses, could overwhelm the border routers and the IXP fabric. In fact, today's IXPs are already vulnerable to high ARP overheads [5].

Fortunately, we can easily reduce the overhead of ARP queries and responses, because each participant needs to learn about only the virtual MAC addresses for its own FECs. As such, the SDX can turn ARP traffic into *unicast* traffic by installing the appropriate rules for handling ARP traffic in switches. In particular, each participant's controller broadcasts a gratuitous ARP response for every virtual next-hop IP address it uses; rules in the IXP's fabric recognize the gratuitous ARP broadcasts and ensure that they are forwarded only to the relevant participant's routers. Participants' routers can still issue ARP queries to map IP addresses to virtual MAC addresses, but the fabric intercepts these queries and redirects them to an ARP relay to avoid overwhelming other routers.

5 Decoupling SDN Policies from Routing

To ensure correctness, any SDX platform must combine SDN policies with dynamic BGP state: which participants have routes to each prefix (*i.e.*, valid next-hop ASes for a packet with a given destination prefix), as well as the next-hop AS to use for each prefix (*i.e.*, the outcome of BGP decision process). The large number of prefixes and participants creates scalability challenges with respect to forwarding table sizes and update rates, before SDN policies even enter the equation.

5.1 Idea: Statically Encode Routing

To reduce the number of rules and updates, we develop a new encoding scheme that is analogous to source routing: The IXP fabric matches on a tag that is provisioned by a participant's SDX controller. To implement this approach, we optimize the tag that the fabric uses to forward traffic (as described in Section 4) to carry information about both the next-hop AS for the packet (as determined by the best BGP route) and the ASes who have advertised routes to the packet's destination prefix. If no SDN policy matches a packet, iSDX can simply match on the next-hop AS bits of the tag to make a default forwarding decision. As before, the sender discovers this tag via ARP.

To implement default forwarding, the IXP fabric maintains static entries for each next-hop AS which forward to participants based upon the next-hop AS bits of the tag. When the best BGP routes change, the entries need not change, rather the next-hop AS bits of the tags change.

To account for changes in available routes, SDN policies that reroute to some participant X confirm whether X has advertised a route before forwarding. The method of checking for X in the tags is static, meaning that in contrast to our previous design [14], BGP updates induce zero updates in the IXP switch data plane. Instead, BGP updates result in tag changes, and the participant's border router learns these dynamic tags via ARP.

5.2 Encoding Next-Hop and Reachability

We now describe how iSDX embeds both the next-hop AS (*i.e.*, from the best BGP route) and the reachability information (*i.e.*, the set of ASes that advertise routes to some prefix) into this tag.

5.2.1 Next-hop encoding

The next-hop information denotes the default next-hop AS for a packet, as determined by BGP. In the example from Section 2.1, A's next-hop AS for traffic to $P1$ as determined by the best BGP route is D. iSDX allocates bits from the tag (*i.e.*, the virtual MAC, which is written into the destination MAC of the packet's header) to denote this next-hop. If no SDN policy overrides this default, iSDX applies a default priority prefix-based match on these bits to direct traffic to the corresponding next-hop.[1] This approach reduces the forwarding table entries in a participant's outbound table, since additional entries for default BGP forwarding no longer need to be represented as distinct entries in the forwarding table. Encoding the next hop information in this way requires $\lg(N)$ bits, where N is the number of IXP participants. At a large IXP with up to 1024 participants, ten bits can encode information about default next-hop ASes, leaving 37 bits.[2]

[1] The OpenFlow 1.3 standard supports this feature [27], which is already implemented in many hardware switches (*e.g.*, [26, 28]).

[2] One of the 48 bits in the MAC header is reserved for multicast.

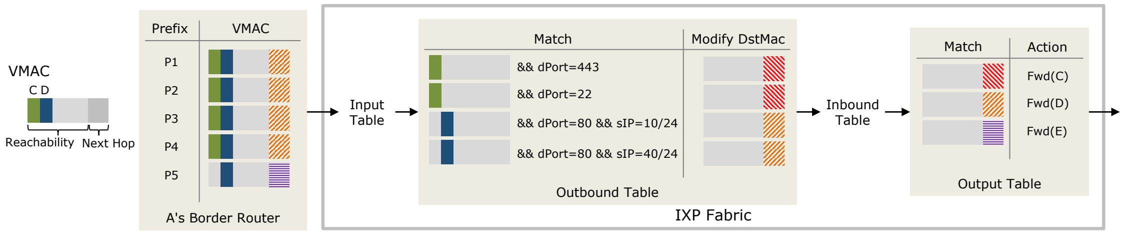


Figure 6: *How AS A's controller uses reachability encoding to reduce the number of flow rules.*

5.2.2 Reachability encoding

We now explain how to encode reachability information into the remaining 37 bits of the destination MAC address. We first present a strawman approach that illustrates the intuition before describing the scalable encoding.

Strawman encoding. Suppose that for a given tag, the i-th bit is 1 if that participant learns a BGP route to the corresponding prefix (or prefixes) via next-hop AS i. Such an encoding would allow the IXP fabric to efficiently determine whether some participant could forward traffic to some next-hop AS i, for any i at the IXP. Considering the example in Section 2.1, A's outbound policies are:

$\texttt{dMac = XX}\mathbf{1}\texttt{X...X} \wedge \texttt{dPort=443} \rightarrow \texttt{fwd}(C)$

$\texttt{dMac = XX}\mathbf{1}\texttt{X...X} \wedge \texttt{dPort=22} \rightarrow \texttt{fwd}(C)$

$\texttt{dMac = XXX}\mathbf{1}\texttt{X..X} \wedge \texttt{dPort=80} \wedge \texttt{sIp=10/24} \rightarrow \texttt{fwd}(D)$

$\texttt{dMac = XXX}\mathbf{1}\texttt{X..X} \wedge \texttt{dPort=80} \wedge \texttt{sIp=40/24} \rightarrow \texttt{fwd}(D)$

where X stands for a wildcard match (0 or 1). This encoding ensures correct interoperation with BGP, yet we use just four forwarding table entries, which is fewer than the 18 required using augmentation (from the original example in Section 2).

Figure 6 explains how this approach reduces the number of forwarding table entries in the switch fabric. When a packet arrives, its virtual MAC encodes both (1) which ASes have advertised a BGP route for the packet's destination ("reachability") and (2) the next-hop participant corresponding to the best BGP route ("next hop"). Suppose that a packet is destined for $P1$ from A; in this case, A's border router will affix the virtual MAC as shown. If that virtual MAC does not match any forwarding table entries in the outbound table, the packet will simply be forwarded to the appropriate default next hop (in this case, D) based on the next-hop encoding. This process makes it possible for the switch to forward default BGP traffic without installing any rules in the outbound table, significantly reducing the size of this table.

Hierarchical encoding. The approach consumes one bit per IXP participant, allowing at most for only 37 IXP participants. To encode more participant ASes in these 37 bits, we divide this bitspace hierarchically. Suppose that an IXP participant has SDN policies that refer to N other IXP participants (*i.e.*, possible next-hop ASes). Then, all of these N participants need to be efficiently encoded in the 37-bit space, B. We aim to create W bitmasks $\{B_1, B_2, \ldots, B_W\}$ that minimize the total number of forwarding table entries, subject to the limitations of the total length of the bitmask.

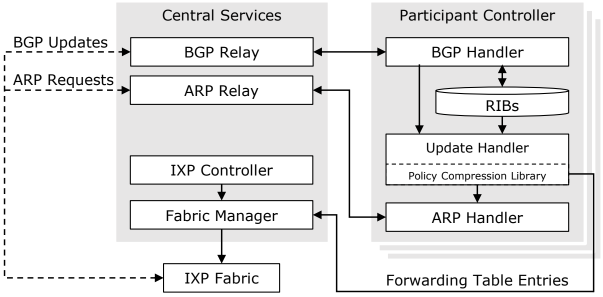


Figure 7: *Implementation of iSDX. It has five main modules: (1) IXP controller, (2) participant SDN controller, (3) ARP relay, (4) BGP relay, and (5) fabric manager.*

Given M prefixes and N IXP participants, we begin with M bitmasks, where each bitmask encodes some *set* of participants that advertise routes to some prefix p_i. We greedily merge pairs of sets that have at least one common participant, and we always merge two sets if one is a subset of the other. Iterating over all feasible merges has worst-case complexity $O(M^2)$; and there may be as many as $M-1$ merge actions in the worst case. Each merge has complexity $O(N)$, which gives us an overall worst-case running time complexity of $O(M^3N)$.

Given 37 spare bits in the destination MAC for reachability encoding, if a participant has defined SDN policies for more that 37 participants who advertise the same prefix, then the number of bits required to encode the reachability information will exceed 37. Our analysis using a dataset from one of the largest IXPs in the world found that the maximum number of participants advertising the same prefix was only 27, implying that largest bitmask that this encoding scheme would require is 27 bits. There were 62 total bitmasks, meaning 6 bits are required to encode the ID of a bitmask, requiring a total of 33 bits for the encoding. Using a different (or custom) field in a packet header might also be possible if these numbers grow in the future.

6 Implementation

We now describe an implementation of iSDX, as shown in Figure 7. Our Python-based implementation has about 5,000 lines of code. Source code and tutorials are publicly available on Github [16]. We have also provided instructions for deploying iSDX on hardware switches [17]; one large government agency has successfully done so with the Quanta LY2 switch [28]. About 300 students used an earlier version of iSDX in the Coursera SDN course [8].

The **fabric manager** is based on Ryu [31]. It listens for forwarding table modification instructions from the participant controllers and the IXP controller and installs the changes in the switch fabric. The fabric manager abstracts the details of the underlying switch hardware and OpenFlow messages from the participant and the IXP controllers and also ensures isolation between participants.

The **IXP controller** installs forwarding table entries in the input and output tables in the switch fabric via the fabric manager. Because all of these rules are static, they are computed only at initialization. Moreover, the IXP controller handles ARP queries and replies in the fabric and ensures that these messages are forwarded to the respective participants' controllers via **ARP relay**.

The **BGP relay** is based on ExaBGP [11] and is similar to a BGP route server in terms of establishing peering sessions with the border routers. Unlike a route server, it does not perform any route selection. Instead, it multiplexes all BGP routes to the participant controllers.

Each **participant SDN controller** computes a compressed set of forwarding table entries, which are installed into the inbound and outbound tables via the fabric manager, and continuously updates the entries in response to the changes in SDN policies and BGP updates. The participant controller receives BGP updates from the BGP relay. It processes the incoming BGP updates by selecting the best route and updating the RIBs. We developed APIs to use either of MongoDB [24], Cassandra [2] and SQLite [34] for storing participants' RIBs. We used the MongoDB (in-memory) for the evaluation in Section 7. The participant controller also generates BGP announcements destined to the border routers of this participant, which are sent to the routers via the BGP relay.

Each participant controller's **update handler** determines whether the inbound and outbound tables need to be updated, as well as whether new gratuitous ARP messages must be sent to the participant's border routers to update any virtual destination MAC addresses. The controller receives ARP requests from the participant's border routers via the **ARP handler** and determines the corresponding ARP reply. The controller also receives SDN policy updates from the network operators in the form of addition and removal lists. Both the update handler and the ARP handler use a policy compression library that we implemented, which provides the mapping between IP prefixes and virtual next-hop IPs (corresponding to best BGP routes), and between virtual next-hop IPs and virtual destination MAC addresses (*i.e.*, an ARP table).

	MDS	NH Encoding	Reachability Encoding
iSDX-D	✓	✗	✗
iSDX-N	✓	✓	✗
iSDX-R	✗	✓	✓

Table 2: *Three distributed SDX Controllers.*

7 Evaluation

We now demonstrate that iSDX can scale to the forwarding table size, data plane update rate, and control plane computation requirements of an industrial-scale IXP. Table 2 summarizes the three different iSDX designs that we compare to previous approaches: iSDX-D applies the same MDS compression technique as in our previous work [14], but with tables distributed across participants; iSDX-N additionally encodes the next-hop AS in the tag; and iSDX-R encodes both the next-hop AS and BGP reachability information in the tag.

Table 3 summarizes our results: *iSDX reduces the number of forwarding table entries for an industrial-scale IXP by three orders of magnitude as compared to an unoptimized, centralized SDX design; and by more than two orders of magnitude over the state-of-the-art SDX design [14].* This section explains these results in detail.

7.1 Experiment Setup

We use data sets from one of the largest IXPs worldwide, which interconnects more than 600 participants, who peer with one another via a BGP route server [29]. We had access to the RIB table dump collected from the IXP's route server on August 8, 2015 for 511 IXP participants. These datasets contain a total of 96.6 million peering (*i.e.*, non-transit) routes for over 300,000 distinct prefixes. We also use a trace of 25,676 BGP update messages from these participants to the route server for the two hours following the collection of this RIB table dump (the participants' RIBs are naturally not perfectly aligned, since dumping a BGP table of about 36 GB from the router takes about fifteen minutes). Our data set does not contain any user data or any personal information that identifies individual users. We run our experiments on a server installed at this IXP configured with 16 physical cores at 3.4 GHz and 128 GB of RAM.

This IXP does not use a programmable IXP fabric, so we assume how participants *might* specify SDN policies, as described in Section 2.1. Specifically, *each participant* has between one to four outbound policies for each of 10% of the total participants. The number of policies and set of participants are chosen uniformly at random. Our sensitivity analysis on this percentage shows that our

			iSDX		
	Unoptimized	Centralized MDS-SDX [14]	iSDX-D	iSDX-N	iSDX-R
Number of Forwarding Table Entries	68,476,528	21,439,540	763,000	155,000	65,250
Policy Compression Time (s)	N/A	297.493	0.0629	0.111	2.810

Table 3: *Summary of evaluation results for iSDX with 500 IXP participants. Note that compression times for iSDX are per-participant, since each participant can compile policies in parallel; even normalizing by this parallelization still yields significant gains.*

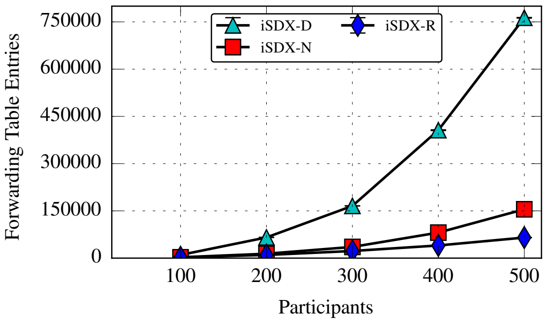


Figure 8: *Number of forwarding table entries.*

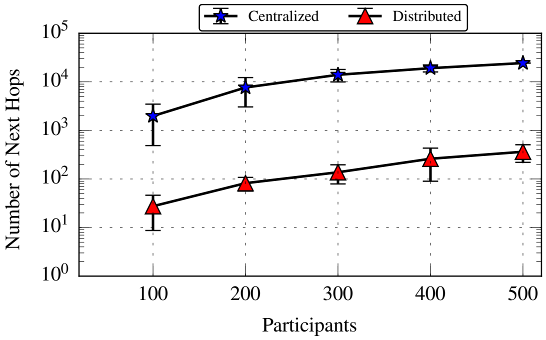


Figure 9: *Number of virtual next-hop IP addresses for centralized and distributed control planes. Results for distributed iSDX do not depend on encoding or compression approach.*

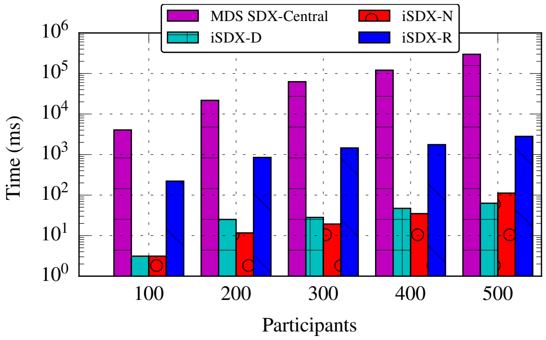


Figure 10: *Time to perform policy compression.*

results are influenced in magnitude but the underlying trends remain. Note that this setup is more taxing than the one in our previous work [14] where only 20% of the total participants had any SDN policies at all. We also evaluate iSDX's performance for smaller IXPs by selecting random subsets of IXP participants (ranging from 100 to 500 ASes) and considering only the RIB information and BGP updates for those participants. We also repeated experiments using public RIB dumps and BGP updates collected by RIPE's RIS servers from 12 other IXPs [30]. As the observed workload was much smaller in this case, we omit these results for brevity.

7.2 Steady-State Performance

We first evaluate the steady-state performance of iSDX. To do so, we use the RIB dumps to initialize the SDX controller (multiple of them for the distributed case) and evaluate the overall performance in terms of the efficiency of data-plane compression, and the time to compile policies and compress them into smaller forwarding tables.

Efficiency of compression. Figure 8 shows the number of forwarding table entries for the three distributed controllers: iSDX-D, iSDX-N, and iSDX-R. The number of forwarding table entries increases with the increasing number of IXP participants. Each of our techniques progressively improves scalability. We observe that the number of forwarding table entries for iSDX-R is very close to the lower bound (*i.e.*, best case), where the number of forwarding table entries is equal to the number of SDN policies.

We also explore the effects of distributing the control plane computation on the ability of iSDX to perform MDS compression. The results are shown in Figure 9. Given 500 participants, partitioning the control plane reduces the number of next hop entries for the border router from 25,000 to 360. This reduction mitigates the load on the border routers, since the number of virtual next hop IP addresses reflects the number of ARP entries each participant's border router must maintain.

Time to perform policy compression. Figure 10 shows the compression time for each controller; this time dominates control-plane computation but only occurs at initialization. The Centralized MDS-SDX operates on a large input rule matrix, and thus requires nearly five minutes to compress policies. iSDX-D distributes the computation across participants, reducing compression time by three orders of magnitude. iSDX-R takes longer than iSDX-D and iSDX-N controllers. For 500 participants, policy compression takes about three seconds.

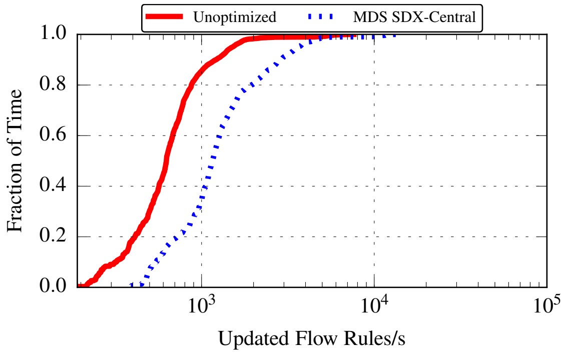


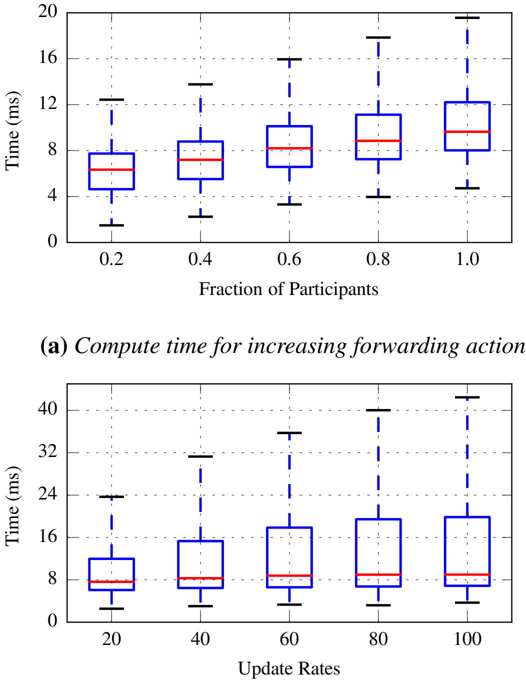
Figure 11: *Rate at which forwarding table entries are updated.*

7.3 Runtime Performance

After iSDX initializes, we replay a two-hour trace of BGP updates from one of the largest IXPs in the world to evaluate the runtime performance of iSDX compared to other SDX designs. We focus on how iSDX reduces the number of forwarding table updates induced by BGP updates and policy changes, as well as the corresponding increase in gratuitous ARP traffic, which is the cost we pay for increased forwarding table stability.

Forwarding table updates in response to routing. Figure 11 shows the cumulative distribution of the number of updated forwarding table entries per second the SDX must process for a BGP update stream coming from all 511 participants at the IXP. MDS compression, which is used in iSDX-D and iSDX-N, significantly increases the rate of updates to the forwarding table in comparison to an unoptimized SDX; this result makes sense because any change to forwarding is more likely to trigger a change to one of the encoded forwarding table entries. *With iSDX-R, there are never updates to the forwarding table entries in response to BGP updates.*

Update latency in response to BGP updates. We aim to understand how quickly iSDX-R can update forwarding information when BGP updates arrive. For iSDX-R, this update time effectively amounts to computing updated virtual next-hop IP and MAC addresses, since iSDX-R never needs to update the IXP fabric forwarding table entries in response to BGP updates. We evaluate update latency with two experiments. First, we vary the fraction of IXP participants to which each IXP participant forwards with SDN policies. For example, if the fraction is 1, each participant has between one and four SDN forwarding policies (at random) for every other SDN participant. Figure 12a shows this result; in all cases, the median update latency in response to a BGP update is less than 10 ms, and the 95th percentile in the worst case is less than 20 ms. Even when we perform simultaneous compilation of all 511 participants on just three servers at the IXP, the median update time is only 52 ms, well within practical requirements.



(a) *Compute time for increasing forwarding actions.*

(b) *Compute time for increasing sustained rates of BGP updates.*

Figure 12: *Latency of iSDX-R updates in response to BGP update streams.*

To understand how iSDX-R behaves when it receives larger update bursts, we evaluate the update latency for increasing sizes of BGP update bursts. We vary the number of BGP updates per second from 20 to 100 and send a constant stream of updates at this rate for five minutes, tracking the latency that the iSDX requires to process the updates. (Although a table reset would presumably cause a very large update burst, the fastest sustained BGP update rate we observed in the trace was only about 35 BGP updates per second.) Figure 12b shows this result. For example, for a rate of 100 BGP updates per second, the median update latency is about 8 ms and the 95th percentile is percentile is about 45 ms.

Gratuitous ARP overhead. Recall that SDX relies on gratuitous ARP to update virtual destination MAC addresses when forwarding behavior changes, often in lieu of updating the forwarding table itself. A centralized SDX control plane sends this ARP response to all IXP participants, but a distributed SDX can send this response only to the border router whose route changed. Figure 13 shows the distribution of the rate at which a participant's border router receives gratuitous ARP messages from the IXP controller in response to BGP routing changes, for both the centralized design (*i.e.*, centralized MDS) and the

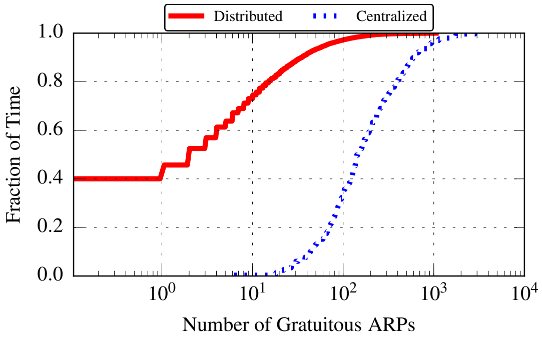


Figure 13: *Rate at which a participant's border router receives gratuitous ARPs.*

distributed one (*i.e.*, iSDX); these rates are independent of which encoding the iSDX uses.

8 Related Work

Ongoing SDX Projects. Software-defined IXPs have been gaining momentum in the past few years [3, 22, 39], and limited real-world deployments are beginning to emerge. Yet, these existing deployments have focused on either smaller IXPs or on forwarding traffic for a partial routing table. Our original SDX [14] work introduced mechanisms for applying SDN policies to control interdomain traffic flow at an IXP and introduced some simple mechanisms for forwarding table compression; yet its capability for compressing and updating forwarding tables cannot meet either the scale or speed demands of the largest industrial IXP. Google's Cardigan SDX controller has been deployed in a live Internet exchange in New Zealand [4, 35]. Cardigan does not use any of the compression techniques that we use in either SDX or iSDX. As a result, we expect that the size of Cardigan's forwarding tables would be similar to the "unoptimized" results that we present in Section 2—orders of magnitude too large for use with hardware switches in large IXPs. Control Exchange Points [21] propose to interconnect multiple SDN IXPs to provide QoS services to the participants and is less concerned with the design of an individual SDN-based IXP.

Distributed SDN controllers. HyperFlow [37], Onix [20], and Devolved Controllers [36] implement distributed SDN controllers that maintain eventually consistent network state partitioning computation across multiple controllers such that each operates on less state. Kandoo [15] distributes the control plane for scalability, processing frequent events in highly replicated local control applications and rare events in a central location. Several distributed controllers focus on fault-tolerance [6, 10, 19]. In contrast to these systems, each participant controller in iSDX operates independently and requires no state synchronization. iSDX's partitioning is first and foremost intended to achieve more efficient compression of forwarding table entries; other benefits, such as parallel computation and fault tolerance, are incidental benefits.

Techniques for data-plane scalability. Other work seeks to address the problem of small forwarding tables in hardware. Data-plane scaling involves (1) rule partitioning [40], where data plane rules are partitioned across multiple switches and incoming traffic is steered to load balance across these switches; and (2) caching [18, 32], which stores forwarding table entries for only a small number of flows in the data plane. These techniques are orthogonal to the compression that iSDX uses. Labeling packets for FIB compression has been applied in various contexts, such as MPLS [9], Fabric [7], LISP [12], and Shadow Macs [1]. These techniques all reduce the number of forwarding table entries in certain routers, often by pushing complex policies to the edge of the network. These techniques generally apply in the wide area, and cannot be directly applied to an IXP topology, although some of the techniques are analogous.

9 Conclusion

Software-Defined Internet Exchange Points (SDXes) are poised to reshape interdomain traffic delivery on the Internet, yet realizing this vision ultimately requires the design and implementation of an SDX that can scale to (and beyond) the largest industrial IXPs on the Internet today. To address this challenge, we developed iSDX, the first SDX controller that scales to large industrial IXPs. We demonstrated how the principles of modularity and decoupling are necessary to scale the control and the data planes. The specific approaches we suggest—partitioning and compression—are applicable in various settings where where composition of forwarding policies is required (*e.g.*, SDN WAN). We have released a public, open-source implementation of iSDX on Github [16], along with tutorials and instructions that have helped catalyze early adoption. Our evaluation shows that iSDX reduces both forwarding table size and the time to compute these entries by several orders of magnitude—enough to make iSDX practical for real operation. Using BGP routing updates from a route server at one of the world's largest IXPs, we showed that iSDX can support industrial-scale operation.

Acknowledgments. We thank our shepherd Dave Oran, Bryan Larish, Jamie Brown, Inder Monga, Rick Porter, Glenn Gardner, Marc Pucci, David Jorm, and the anonymous reviewers for the feedback and comments. This research was supported by National Science Foundation Awards CNS-1539920, CNS-1409056, and a research contract with the Laboratory for Telecommunications Sciences. This research was also supported by European Union's Horizon 2020 research and innovation program under the ENDEAVOUR project (grant agreement 644960).

References

[1] Agarwal, K., Dixon, C., Rozner, E., and Carter, J. Shadow macs: Scalable label-switching for commodity ethernet. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking* (New York, NY, USA, 2014), HotSDN '14, ACM, pp. 157–162. (Cited on page 12.)

[2] Apache Cassandra. http://cassandra.apache.org/. (Cited on page 9.)

[3] AtlanticWave-SDX. https://itnews.fiu.edu/wp-content/uploads/sites/8/2015/04/AtlanticWaveSDX-Press-Release_FinalDraft.pdf. (Cited on page 12.)

[4] Bailey, J., Pemberton, D., Linton, A., Pelsser, C., and Bush, R. Enforcing rpki-based routing policy on the data plane at an internet exchange. HotSDN, ACM. (Cited on pages 1 and 12.)

[5] Boteanu, V., Bagheri, H., and Pels, M. Minimizing arp traffic in the ams-ix switching platform using openflow. (Cited on page 7.)

[6] Canini, M., Kuznetsov, P., Levin, D., and Schmid, S. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *INFOCOM* (2015). (Cited on page 12.)

[7] Casado, M., Koponen, T., Shenker, S., and Tootoonchian, A. Fabric: A retrospective on evolving sdn. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks* (New York, NY, USA, 2012), HotSDN '12, ACM, ACM, pp. 85–90. (Cited on page 12.)

[8] Coursera SDN Course, 2015. https://www.coursera.org/course/sdn1. (Cited on page 9.)

[9] Davie, B. S., and Rekhter, Y. *MPLS: technology and applications*. San Francisco, 2000. (Cited on page 12.)

[10] Dixit, A., Hao, F., Mukherjee, S., Lakshman, T., and Kompella, R. Towards an elastic distributed sdn controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (New York, NY, USA, 2013), HotSDN '13, ACM, ACM, pp. 7–12. (Cited on page 12.)

[11] ExaBGP. https://github.com/Exa-Networks/exabgp. (Cited on page 9.)

[12] Farinacci, D., Fuller, V., Meyer, D., and Lewis, D. The locator/id separation protocol (lisp). Internet Requests for Comments, January 2013. http://www.rfc-editor.org/rfc/rfc6830.txt. (Cited on page 12.)

[13] Feamster, N., Rexford, J., Shenker, S., Clark, R., Hutchins, R., Levin, D., and Bailey, J. Sdx: A software defined internet exchange. *Open Networking Summit* (2013). (Cited on page 1.)

[14] Gupta, A., Vanbever, L., Shahbaz, M., Donovan, S. P., Schlinker, B., Feamster, N., Rexford, J., Shenker, S., Clark, R., and Katz-Bassett, E. SDX: A Software Defined Internet Exchange. In *ACM SIGCOMM* (Chicago, IL, 2014), ACM, pp. 579–580. (Cited on pages 1, 2, 3, 4, 6, 7, 9, 10 and 12.)

[15] Hassas Yeganeh, S., and Ganjali, Y. Kandoo: A framework for efficient and scalable offloading of control applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks* (New York, NY, USA, 2012), HotSDN '12, ACM, ACM, pp. 19–24. (Cited on page 12.)

[16] iSDX Gihub Repo. https://github.com/sdn-ixp/iSDX. (Cited on pages 2, 9 and 12.)

[17] iSDX HW Test Instructions. https://github.com/sdn-ixp/iSDX/tree/master/examples/test-ms/ofdpa. (Cited on page 9.)

[18] Katta, N., Alipourfard, O., Rexford, J., and Walker, D. Infinite cacheflow in software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking* (New York, NY, USA, 2014), HotSDN '14, ACM, ACM, pp. 175–180. (Cited on page 12.)

[19] Katta, N., Zhang, H., Freedman, M., and Rexford, J. Ravana: Controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (New York, NY, USA, 2015), SOSR '15, ACM, pp. 4:1–4:12. (Cited on page 12.)

[20] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., and Shenker, S. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6. (Cited on page 12.)

[21] Kotronis, V., Dimitropoulos, X., Klöti, R., Ager, B., Georgopoulos, P., and Schmid, S. Control exchange points: Providing qos-enabled end-to-end services via sdn-based inter-domain routing orchestration. (Cited on page 12.)

[22] LightReading. Pica8 Powers French TOUIX SDN-Driven Internet Exchange, June 2015. http://ubm.io/1Vc0SLE. (Cited on pages 1 and 12.)

[23] Mambretti, J. Software-defined network exchanges (SDXs) and software-defined infrastructure (SDI), June 2014. Presentation at the Workshop on Prototyping and Deploying Experimental Software Defined Exchanges (SDXs). (Cited on page 1.)

[24] mongoDB. https://www.mongodb.org/. (Cited on page 9.)

[25] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 1–14. (Cited on page 6.)

[26] NoviSwitch 1132. http://noviflow.com/wp-content/uploads/2014/09/NoviSwitch-1132-Datasheet.pdf. (Cited on pages 4, 5 and 7.)

[27] Openflow 1.3 specifications. http://bit.ly/1eyrkxY. (Cited on page 7.)

[28] QuantaMesh BMS T3048-LY2. http://www.qct.io/Product/Networking/Bare-Metal-Switch/QuantaMesh-BMS-T3048-LY2-p55c77c75c159. (Cited on pages 7 and 9.)

[29] Richter, P., Smaragdakis, G., Feldmann, A., Chatzis, N., Boettger, J., and Willinger, W. Peering at peerings: On the role of ixp route servers. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 31–44. (Cited on page 9.)

[30] RIPE. Ris raw data, 2015. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data. (Cited on page 10.)

[31] Ryu SDN Framework. http://osrg.github.io/ryu/. (Cited on pages 1 and 9.)

[32] Sarrar, N., Uhlig, S., Feldmann, A., Sherwood, R., and Huang, X. Leveraging zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review 42*, 1 (January 2012), 16–22. (Cited on page 12.)

[33] Smolka, S., Eliopoulos, S., Foster, N., and Guha, A. A Fast Compiler for NetKAT. In *ICFP* (2015). (Cited on page 6.)

[34] SQLite. https://www.sqlite.org/. (Cited on page 9.)

[35] Stringer, J., Pemberton, D., Fu, Q., Lorier, C., Nelson, R., Bailey, J., Correa, C., and Esteve Rothenberg, C. Cardigan: Sdn distributed routing fabric going live at an internet exchange. In *Computers and Communication (ISCC), 2014 IEEE Symposium on* (June 2014), IEEE, pp. 1–7. (Cited on pages 1, 2 and 12.)

[36] Tam, A.-W., Xi, K., and Chao, H. Use of devolved controllers in data center networks. In *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on* (April 2011), IEEE, pp. 596–601. (Cited on page 12.)

[37] Tootoonchian, A., and Ganjali, Y. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking* (Berkeley, CA, USA, 2010), INM/WREN'10, USENIX Association, USENIX Association, pp. 3–3. (Cited on page 12.)

[38] Whyte, S. Project CARDIGAN An SDN Controlled Exchange Fabric. https://www.nanog.org/meetings/nanog57/presentations/Wednesday/wed.lightning3.whyte.sdn.controlled.exchange.fabric.pdf, 2012. (Cited on page 4.)

[39] Workshop on Prototyping and Deploying Experimental Software Defined Exchanges. https://www.nitrd.gov/nitrdgroups/images/4/4d/SDX_Workshop_Proceedings.pdf. (Cited on page 12.)

[40] Yu, M., Rexford, J., Freedman, M. J., and Wang, J. Scalable flow-based networking with difane. *SIGCOMM Computer Communication Review 40*, 4 (August 2010), 351–362. (Cited on page 12.)

XFabric: A Reconfigurable In-Rack Network for Rack-Scale Computers

Sergey Legtchenko *Microsoft Research*
Nicholas Chen *Microsoft Research*
Daniel Cletheroe *Microsoft Research*
Antony Rowstron *Microsoft Research*
Hugh Williams *Microsoft Research*
Xiaohan Zhao* *Microsoft Research*

Abstract

Rack-scale computers are dense clusters with hundreds of micro-servers per rack. Designed for data center workloads, they can have significant power, cost and performance benefits over current racks. The rack network can be distributed, with small packet switches embedded on each processor as part of a system-on-chip (SoC) design. Ingress/egress traffic is forwarded by SoCs that have direct uplinks to the data center. Such fabrics are not fully provisioned and the chosen topology and uplink placement impacts performance for different workloads.

XFabric is a rack-scale network that reconfigures the topology and uplink placement using a circuit-switched physical layer over which SoCs perform packet switching. To satisfy tight power and space requirements in the rack, XFabric does not use a single large circuit switch, instead relying on a set of independent smaller circuit switches. This introduces partial reconfigurability, as some ports in the rack cannot be connected by a circuit. XFabric optimizes the physical topology and manages uplinks, efficiently coping with partial reconfigurability. It significantly outperforms static topologies and has a performance similar to fully reconfigurable fabrics. We demonstrate the benefits of XFabric using flow-based simulations and a prototype built with electrical crosspoint switch ASICs.

1 Introduction

There is a trend in large-scale data centers towards higher per-rack server density. While a typical compute rack today is composed of 40 to 50 blade servers interconnected through a Top of Rack (ToR) switch, hardware vendors increasingly propose energy-efficient, high density micro-servers, designed for data center workloads [17, 27, 35]. Rack-scale computers, such as AMD SeaMicro [44], HP Moonshot [1] and Boston Viridis [8] are up to rack-scale high density clusters of micro-servers with tight integration of the network, storage and compute. For example, the Boston Viridis supports hundreds of SoCs in one standard data center rack. Rack-scale computers are optimized for commodity data center workloads and have significant power, cost and performance benefits over traditional racks [4, 21, 22, 38] and attract increasing research interest [5, 6, 14, 16, 39, 42].

Higher server density requires a redesign of the in-rack network. A fully provisioned 40 Gbps network with 300 SoCs would require a ToR switch with 12 Tbps of bisection bandwidth within a rack enclosure which imposes power, cooling and physical space constraints. For example, the peak power draw (for power, compute, storage and networking) is limited by the power distribution used in data centers and the amount of heat that the in-rack cooling is able to dissipate and is around 9-16 kW for a typical high density rack today [3]. To address these challenges, some proposed designs replace a ToR switch by a "distributed fabric" where the packets forwarding is done by the servers. If the system uses SoCs then a small packet switch can be embedded on the server's SoC. For example, Boston Viridis uses the Calxeda EnergyCore SoC which has an embedded packet switch supporting eight 10 Gbps lanes. Each SoC is connected to a subset of SoCs in the rack, forming a multi-hop, bounded degree topology, e.g. mesh or torus. Each SoC forwards in-rack traffic from other SoCs and ingress/egress traffic is tunneled through a set of SoCs that have direct uplinks to the data center network.

Distributed fabrics are cost effective, but lack the flexibility of a fully provisioned network. Bisection bandwidth and end-to-end latency in the rack are a function of the network topology, and the best topology depends on the expected workload. Ingress/egress traffic is forwarded through multiple hops in the rack to an uplink, interfering with in-rack traffic. Lack of flexibility leads to suboptimal performance and complicates the design. For example, the HP Moonshot has three independent

* while on internship from UCSB.

networks with different topologies within the same enclosure: a *radial fabric* for ingress/egress traffic, and multi-hop *storage* and *2D torus* fabrics for in-rack traffic.

XFabric is a rack-scale network that maintains the benefits of a distributed fabric but allows workload-specific reconfigurability of the topology and uplinks. XFabric is organized as a *packet-switched* network running over a *physical circuit-switched* network that allows the physical topology of the fabric to be dynamically reconfigured. This could be achieved by using a single large circuit switch that would provide *full reconfigurability*, so *any* two SoC ports in the rack can be directly connected. However, XFabric needs to operate within the space and power limitations of the rack.

To achieve this, XFabric uses *partial reconfigurability*. It partitions the physical layer into a *set* of smaller independent circuit switches such that each SoC has a port attached to each partition. Packets can be routed between the partitions by the packet switches embedded in the SoCs. The partitioning significantly reduces the circuit switch port requirements enabling a single crosspoint switch ASIC to be used per partition. This makes XFabric deployable in a rack at reasonable cost.

However, the challenge is that the fabric is no longer fully reconfigurable, as SoC ports attached to different crosspoint switch ASICs cannot be connected directly. XFabric uses a novel topology generation algorithm that is optimized to generate a topology and determine which circuits should be established per partition. It also generates the appropriate forwarding tables for each SoC packet switch. The algorithm is efficient, and XFabric can instantiate topologies frequently, e.g. every second at a scale of hundreds of SoCs, if required. Additionally, it is able to place uplinks to the data center enabling them to be efficiently reconfigured.

XFabric uses insights from extensive work on reconfigurable data center-scale networks that enable dynamically reconfigurable network links between ToR switches [13, 20, 23, 24, 41, 47]. Similar to prior work, e.g. OSA [13] and FireFly [24], the topology is reconfigured at the physical layer, and network traffic is forwarded through multiple hops over the reconfigurable topology. XFabric differs in that it has been designed to operate at a rack-scale with SoCs that have embedded packet switches with multiple ports. It neither relies on wireless technology that cannot be used in the rack, nor requires a single large circuit switch. Designed for cost-effective in-rack deployments, XFabric sacrifices full reconfigurability for partial reconfigurability and demonstrates that this still provides good performance.

We have a prototype cluster, which uses 27 servers emulating SoCs and an XFabric network built with custom 32-port switches using low cost commodity crosspoint switch ASICs. We evaluate XFabric using this prototype and a flow-based simulator at larger scale. The results show that under realistic workload assumptions, the performance of XFabric is up to six times better than a static 3D-Torus topology at rack scale. We also show it provides comparable performance to a fully reconfigurable network while consuming five times less power.

The rest of the paper is organized as follows. Sec. 2 motivates our design and Sec. 3 provides an overview of XFabric. Sec. 4 details the algorithms used by the controller. Sec. 5 describes our current implementation of XFabric. Sec. 6 evaluates the performance of XFabric. Finally, Sec. 7 and 8 present related work and conclude.

2 Partial Reconfigurability

Reconfigurable networks have been traditionally proposed at data center scale [13, 20, 23, 24, 41, 47]. In these networks, each ToR has d reconfigurable ports and the set of d ToRs to which each ToR is directly connected is dynamically adapted to match the traffic demand. This has been implemented either with wireless (e.g. RF-based or free-space optics) [23, 24] or Optical Circuit Switches (OCS) [13, 20, 41, 47]. In the latter case, all d ports of all the n ToRs are connected to the same OCS that acts as a circuit switch with $n \times d$ ports: *any* port can be connected to *any* port of *any* ToR and the network topology is *fully reconfigurable*.

XFabric is focused on providing reconfigurability at the rack-scale, which has unique challenges because of the additional constraints due to physical space, power and cooling limitations. At the densities that can potentially be achieved using SoCs, the number of ports is high. If the switch functionality is distributed across the SoCs and a distributed network fabric is used, the number of ports required will be even higher. For example, a fully reconfigurable distributed fabric with 256 SoCs and 6 ports per SoC for in-rack communication requires 1,536 ports. This port count is too high to use a single crosspoint switch ASIC. It is possible to build a circuit switch implemented as a folded Clos network with multiple crosspoint switches, however, a folded Clos to support $n \times d$ ports requires $5 \times n \times d$ ports to be provisioned [13], which, for this example setup would require $5 \times 256 \times 6 = 7,680$ ports.

Fitting this in the rack can be challenging, but powering and cooling it will be hard. The per-port power draw ranges from 0.14 W for a typical optical circuit switch [11] to 0.28 W for a 10 Gbps electrical circuit switch [12]. The switches would consume between 1.3 to 2.6 kW, representing a significant fraction of the power provisioned for a high density rack today [3]. Given that as we increase density the compute and storage power requirements will also increase we need to manage power

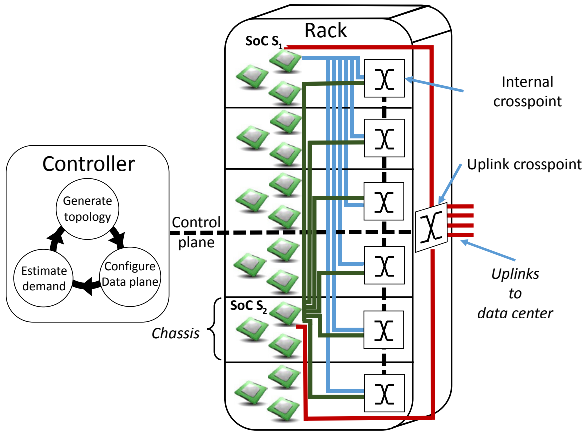


Figure 1: XFabric architecture, $d = 6$, $l = 4$ and $c = 6$.

resources carefully. A fully reconfigurable fabric is probably not acceptable.

In the rest of the paper, we refer to a circuit switch that is of a scale that can be implemented using a single crosspoint switch ASIC as simply a *crosspoint switch*, while one that is implemented as a folded Clos network of multiple crosspoint switches as a *Clos circuit switch*.

XFabric exploits the observation that full reconfigurability is not necessary. XFabric provides a *partially* reconfigurable fabric in which each SoC port can be connected to a *subset* of the SoC ports in the rack. XFabric has d independent physical networks each with a single circuit switch. Each SoC has a port attached to each of the d networks. This means that each SoC port can be connected to any other SoC. Hence, for 256 SoCs with 6 ports per SoC we would need 6 crosspoint switches each with 256 ports. Currently, commodity 160-port electrical crosspoint switches capable of switching 10 Gbps links are available [12]. We believe that a single crosspoint switch ASIC could be built to support approximately 350 ports. This is compared to requiring 7,680 ports for a full folded Clos circuit switch at this scale, requiring 22 crosspoint switch ASICs at 350 ports per ASIC.

Partial reconfigurability performs better in terms of cost and power. In terms of power, if we used a 256-port electrical crosspoint switch then for 256 SoCs with six 10 Gbps ports, XFabric would require 0.4 kW versus a fully reconfigurable fabric using a folded Clos circuit switch that would require 2.2 kW. In terms of cost, the per port cost is approximately \$3, hence XFabric would have a cost of about \$4.6K, while a fully reconfigurable fabric would cost \$23K.

Partial reconfigurability limits the physical network topologies that can be instantiated, which potentially impacts performance. In the next section we describe XFabric in detail, and in Section 6 empirically show that the impact on performance is minimal.

3 XFabric Architecture Overview

The XFabric architecture combines a packet-switched layer 2 operating over a circuit-switched layer 1. It is assumed that each SoC has an embedded packet switch and exposes d ports for internal rack communication e.g. $d = 6$ [46]. Each of these ports is reconfigurable and is connected to a crosspoint switch called an *internal crosspoint*. Figure 1 shows the architecture, with $d = 6$. There are six internal crosspoints, and we highlight for SoCs S_1 and S_2, the links to each of the six internal crosspoints, which could be done on printed circuit board (PCB).

Each rack has l uplinks from the rack to the data center network to carry traffic for destinations outside the rack. The value of l is a function of the expected use for the rack; the Boston Viridis chassis has four 10 Gbps uplinks for 48 SoCs [8]. Each SoC has one *uplink port* to handle ingress/egress traffic to destinations outside the rack in addition to the d internal rack communication ports. Each SoC uplink port is connected to an *uplink crosspoint* which has a set of ports connected to the data center network. For example, in Figure 1 there are 4 uplinks. In operation, the SoCs that do not have their uplink port connected to one of the l external links tunnel their ingress/egress traffic to a SoC which is connected.

XFabric ensures that all d ports on each of the n SoCs are connected to other SoCs and all the l uplinks are connected to SoC uplink ports. We assume that each crosspoint has s ports, and that $s = n$. Due to the fact that a crosspoint with n SoCs connected can establish $n/2$ circuits, we assume that n is even. While our design is not fundamentally restricted to electrical circuit switching, this technology offers several benefits. Crosspoint ASICs are commodity low cost components [12, 50, 54] which are compact and have a low reconfiguration latency. For example, the M21605 crosspoint switch ASIC has 160 12.5 Gbps ports, is available in a 45 mm package and has a maximum reconfiguration latency of only 70 ns [12]. Uplink crosspoints connect SoCs' uplink ports directly to the data center network which requires the uplink crosspoint to support the PHY used outside the rack. The d internal ports can use the same or different PHY depending on the SoC implementation. It could be standard, e.g. backplane Ethernet [53] or proprietary [46], for example to reduce power consumption.

At layer 2, packets are forwarded by the SoCs over the instantiated physical topology using multi-hop routing. Packet switching operates independently from circuit switching, i.e. circuits are not established on a per-packet or per-flow basis. The physical topology reconfiguration is performed every interval t, where t is in the order of seconds. This removes the XFabric reconfiguration logic from the data path, simplifying the design and is motivated by the observation that circuits only need to be reconfigured when the workload traffic pattern changes sufficiently to make reconfiguration beneficial. Layer 2 packet switching over layer 1 circuit switching forms the *data plane* of XFabric.

XFabric is managed by an in-rack controller that receives from each SoC estimates of its traffic demand to other SoCs and the uplink. Figure 1 shows the workflow of the controller. Periodically, it aggregates the information received from the SoCs into a rack-scale demand matrix and computes a new topology optimized for the demand. It then instantiates the topology in the data plane by establishing new circuits at the physical layer and updates the layer 2 forwarding tables. We assume that the packet switches on the SoCs support functionality to allow them to program their forwarding tables, e.g. OpenFlow [40]. The topology generation algorithm is lightweight and operates within the limitations imposed by the partially reconfigurable fabric, only producing topologies that can be instantiated by the network.

The SoC on which the controller executes needs to be connected to a micro-controller associated with each crosspoint ASIC through a *control plane* shown in dotted lines in Figure 1. Our current prototype supports Ethernet and USB control planes and we assume that a small fraction (e.g. 3) have their uplink ports connected to this network rather than an uplink crosspoint. The controller is designed to use only soft state and the reconfiguration process is resilient to the failure of the controller. If the controller fails then the network will be left in a consistent state and the controller can be started on another SoC which is connected to the control plane.

4 XFabric Configuration

XFabric needs to determine the mapping of the uplinks to SoCs and the internal fabric topology. The uplink mapping is performed first, because ingress/egress traffic induces load on the internal fabric while routed to the SoCs with external uplinks. Before describing the uplink mapping and internal topology generation algorithms, we describe how XFabric estimates the traffic demand.

4.1 Traffic Demand Estimation

For internal traffic, each SoC maintains a vector of length n and records the total number of bytes sent to each SoC in the rack. For external traffic, the SoC maintains two values, T_i and T_e, the total number of bytes sent and received, respectively. Periodically, this information is sent to the controller through the data plane and the counters are reset, and we call these *demand vectors*.

The controller maintains two vectors v_i and v_e of size n for ingress/egress traffic in which $v_i[S]$ is the number of bytes sent and $v_e[S]$ the number of bytes received by S during the interval. The controller aggregates the demand vectors into an $n \times n$ demand matrix, dm, such that $dm[S_1,S_2]$ represents a demand weight from S_1 to S_2, maintained using a weighted average.

4.2 Uplink Configuration Algorithm

The uplink configuration selects l SoCs that will be directly connected to the data center network and to which other SoCs need to tunnel their external traffic.

Conceptually, the controller partitions the n SoCs in the rack into l sets and places an uplink on one of the SoCs in each set. This SoC acts as a gateway to the data center network for the rest of the SoCs in the set. The controller aims to balance traffic between uplinks using the demand vectors v_i and v_e. Ideally, the aggregate external traffic demand is the same across all sets and for each set, the uplink is placed on the SoC with heaviest external traffic demand.

The placement algorithm operates in two stages. First, for each of the l uplinks, it selects the SoC S that has the highest demand $D_{ext}[S] = v_i[S] + v_e[S]$ and no uplink and places the uplink on S. In the second stage, the algorithm determines the sets of SoCs associated to each uplink. This is done by ordering SoCs without uplinks by their D_{ext} and iteratively assigning the SoC with highest demand to the set with the least aggregate demand. Ordering SoCs by demand ensures that SoCs with high demand will be fairly balanced across sets. Once all the SoCs have been assigned, source and destination SoCs for all external traffic are known. Based on this knowledge, the algorithm builds a traffic matrix dm_u in which $dm_u[S_1,S_2]$ is the ingress (and $dm_u[S_2,S_1]$ the egress) traffic demand between a SoC S_1 and its uplink placed on S_2. The algorithm then creates a matrix dm_{all} which is a sum of dm_{ext} and dm. This matrix is used by the topology generation algorithm to optimize the in-rack topology to both internal and external traffic.

4.3 Topology Generation Algorithm

This phase computes a topology optimized for dm_{all} by reducing the hop count between SoCs with high demand.

Forwarding high bandwidth traffic through multiple hops consumes bandwidth per link and incurs load on each SoC packet switch it traverses. Lower hop count thus results in lower link load and less resources spent on forwarding, improving network goodput [13]. For latency sensitive traffic, such as in-memory storage using RDMA [18], reducing the round trip time is important. A one hop latency of 1 microsecond versus a four hop latency of 4 microseconds is significant.

Conceptually, for each pair of SoCs in the rack, the algorithm assigns a weight based on their relative demand.

```
Input:
    $socs \leftarrow SoC\_list[n]$
    $dm_{all} \leftarrow demand\_matrix$
    $port\_map \leftarrow XbarToSoCPortMapping[c]$
Output:
    $PacketForwardingTables$
    $CircuitAssignment\ circuits[c]$
1  $topo \leftarrow Disconnected\_Topology(socs)$
2  $SoC\_pairs \leftarrow Order\_By\_Demand(socs, dm_{all})$
3  $xbar\_map \leftarrow To\_Xbar(SoC\_pairs, port\_map)$
4  while $SoC\_pairs \neq \varnothing$ do
5      $partition\_count \leftarrow 0$
6      foreach $soc$ in $socs$ do
7          $part[soc] = \{soc\}$
8          $partition\_count \leftarrow partition\_count + 1$
9      foreach $pair$ in $SoC\_pairs$ do
10         if $part[pair.src] \neq part[pair.dest]$ then
11             $xbars \leftarrow xbar\_map[pair]$
12             $xbar \leftarrow Best\_Ranked(xbars, SoC\_pairs)$
13             $xbars[xbar].Add\_Circuit(\{pair.src, pair.dest\})$
14             $topo.Add\_Undirected\_Edge(pair.src, pair.dest)$
15             $Merge(part[pair.src], part[pair.dest])$
16             $partition\_count = partition\_count - 1$
17             foreach $p$ in $SoC\_pairs$ do
18                 if $xbar\_map.Conflict(p, pair, xbar)$ then
19                     $xbar\_map[p].Remove(xbar)$
20                     if $xbar\_map[p] = \varnothing$ then
21                         $xbar\_map.Remove(p)$
22                         $SoC\_pairs.Remove(p)$
23                     else if $p = pair$ then
24                         $SoC\_pairs.Reinsert(p, p.demand)$
25             if $partition\_count = 1$ then
26                 break
27 return $\{topo.ComputeForwardingTables(),\ circuits\}$
```

Algorithm 1: XFabric topology generation algorithm.

It then iteratively computes disjoint maximum weight spanning trees until all SoC ports in the rack have been assigned. The resulting topology is a union of maximum weight spanning trees and has three key properties. First, by construction it is fully connected, *i.e.,* there exists a path between each pair of SoCs. Second, it maximizes resource usage as all ports are assigned. Finally, as spanning trees are of maximum weight, SoC pairs with heavy traffic demand are satisfied in priority. A key challenge is to support partial reconfigurability and to do this within the constraints imposed by the physical topology.

Algorithm 1 describes the process in detail. The algorithm inputs are a list of SoCs and crosspoint ports to which each is attached (*socs* and *port_map*, which are initialized at boot time) and the demand matrix dm_{all}. It starts by initializing three data structures: *topo*, *SoC_pairs* and *xbar_map* (lines 1 to 3). The first is a fully disconnected topology in which each SoC in the rack is represented by a vertex and to which edges will be greedily added. We define the demand between a pair of SoCs $\{S_1, S_2\}$ as $D_{\{S_1,S_2\}} = dm_{all}[S_1][S_2] + dm_{all}[S_2][S_1]$ and *SoC_pairs* is a list of all pairs of SoCs that can be connected through each crosspoint, ordered by their demands in descending order (highest first). The last is a dictionary that associates each pair of SoCs to a set of crosspoints through which they can be connected. Initially, each pair of SoCs can be connected through any of the d crosspoints.

The main loop (lines 4 to 26) performs a sequence of spanning tree computations and stops when no more SoC pairs can be connected (line 4). The maximum weight computation is based on Kruskal's algorithm [32]: it starts with a set of partitions, one for each SoC (lines 5 to 8), and greedily reduces the number of partitions by connecting the two SoCs that are not in the same partition and have the highest demand (line 10). This results in two partitions being merged as their SoCs are no longer disconnected (lines 15-16). If only one partition remains, all SoCs are connected by a maximum weight spanning tree (line 25-26).

In order to connect a pair of SoCs, the algorithm selects one of the crosspoints through which the connection can be made (lines 11-12). Crossbar ports cannot be reused for multiple circuits simultaneously, therefore connecting a pair of SoCs $\{S_1,\ S_2\}$ through a crosspoint C implies that S_1 or S_2 can no longer be connected to other SoCs through C. It means that establishing a circuit in C negatively impacts its ability to satisfy remaining demand. In order to select a crosspoint in which connecting S_1 to S_2 has the least negative impact, a ranking between the crosspoints is performed (line 12). The ranking function computes the aggregate demand of all connections between S_1, S_2 to any other SoC that has a free port in C. This represents the demand that C would not be able to satisfy if $\{S_1,\ S_2\}$ was established, hence the crosspoint with the lowest value is selected. At that point, both the pair of SoCs and the crosspoint have been determined and the corresponding undirected edge and circuit are added (lines 13-14). Finally, the algorithm updates *SoC_pairs* and *xbar_map* (lines 17 to 24). It removes C from all the pairs in *xbar_map* that can no longer be connected through C (line 18-19). If the SoC pair can no longer be connected through any of the crosspoints, it is removed from *SoC_pairs* (lines 21 to 22). As two SoCs can be connected through multiple crosspoints at the same time, the pair that has just been connected is not removed from the *SoC_pairs* but reinserted after the last pair that has some unsatisfied demand (line 24). That way, if all pairs with demand have been connected, the algorithm can add secondary direct connections between high demand pairs, increasing the bandwidth.

The algorithm executes in polynomial time and once all circuits have been assigned, the topology is optimized for dm_{all}. The algorithm has the property that in addition to computing an optimized topology, it also finds the circuit assignment that instantiates that topology in the partially reconfigurable fabric. The result of the algorithm is a set of forwarding tables derived from the computed topology and the circuit assignment that is merged with the uplink circuit assignment. The controller uses this information to reconfigure the data plane.

4.4 Reconfiguration

To instantiate a new topology, the XFabric controller needs to update the circuit switches (layer 1) and ensure all forwarding tables in each SoC are updated (layer 2). This cannot be achieved instantaneously, and can lead to instability during the update interval. The goal of reconfiguration is to minimize this window of instability.

We considered two general approaches. Inspired by SWAN [25], we experimented with incrementally changing the physical topology to ensure that packets can be successfully routed. This requires identifying a set of intermediate topologies, and then moving traffic off links that are to be reconfigured and then stepping through multiple different intermediate configurations. This approach leads to larger reconfiguration periods: the time taken to reconfigure is approximately constant and independent of the number of links being reconfigured, so migrating through x intermediate topologies takes x times the reconfiguration delay. Hence, we adopted the approach of performing a single reconfiguration.

Before triggering the reconfiguration, the controller sends new circuit assignments to every circuit switch through the control plane and new forwarding state to the SoCs through the data plane. Each circuit switch receives a `map` packet composed of a list of port mappings and a bitmap to indicate which ports need to be reconfigured. The micro-controller on the switch loads the circuit assignments into a set of registers on the crosspoint ASIC and acknowledges the controller, but does not instantiate the circuits. The physical topology must remain unchanged at this stage as the controller has no out-of-band mechanism to communicate with the SoCs. Each SoC receives its new forwarding tables together with the MAC address of the SoC that will be connected to each of its ports and a unique 64-bit version number for the configuration. This is efficiently encoded so the forwarding table, plus all the other information for a XFabric with 512 SoCs is less than 1 KB. Each SoC runs a process that receives and stores the update, but again does not reprogram any forwarding tables. Once all the SoCs have acknowledged the update information, all circuit switches and SoCs are ready for the reconfiguration.

The controller triggers the reconfiguration by transmitting a `reconfigure` packet to each circuit switch through the control plane. When received, the micro-controller on the circuit switch reprograms its crosspoint ASIC to the configuration specified in the `map`. At this point the physical network (layer 1) has been reconfigured, but the forwarding tables at the SoCs have not yet been updated. Once every circuit switch has acknowledged, the controller uses a simple flood-based mechanism to trigger the use of the new forwarding tables on each SoC. It sends on each of its ports a reconfiguration message which includes the new configuration version number. If a SoC with an old forwarding table receives the reconfiguration message, it starts using the new one and issues a reconfiguration message on all its ports. SoCs with new forwarding tables ignore reconfiguration messages. This process ensures rapid reconfiguration, in the worst case the number of rounds will be the diameter of the network.

It is essential that the data plane rapidly converges to a consistent state, even if the controller fails during the process. In particular, if the failure occurs after sending out a `reconfigure` to a subset of the circuit switches, the physical topology could be left in an inconsistent state. To address this, we ensure that each circuit switch that receives a `reconfigure` and has not yet reconfigured broadcasts the message through the control plane.

A failure of the controller before the broadcast of the reconfiguration in the data plane could lead to stale forwarding state at layer 2. To avoid this, we allow the SoCs to locally trigger the update of the forwarding tables. It does this by monitoring the local MAC addresses of SoCs attached to its ports: if a packet is received from a different MAC address than expected the SoC flushes the current forwarding table and uses the new one. After this local update, the SoC broadcasts a reconfiguration message, ensuring that the new forwarding state is propagated despite the failure of the controller.

During the reconfiguration of the switches any packets in flight at the switch can be corrupted or lost. However, thanks to the low switching latency of electrical crosspoint ASICs (see Section 3), we observe the packet loss to be low in practice (see Section 6) and rely on end-to-end transport protocols to recover from the packet loss.

5 XFabric Implementation

We have built a prototype XFabric platform, consisting of a set of seven electrical circuit switch units and 27 servers, each configured with eight 1 Gbps Ethernet NICs and a single Intel Xeon E5520 8-core 2.27GHz CPU running Windows Server 2008 R2 Enterprise. Each server emulates a SoC with an embedded packet switch that has six NIC ports for in-rack traffic. One NIC port is used as an uplink port and the last port is connected to a ToR switch for debug and experiment control.

Each circuit switch unit has 32 ports and each server is connected to all 7 circuit switches. Six serve as internal crosspoints and the last one is an uplink crosspoint with four uplinks ($l = 4$). The switches use the Analog Devices ADN4605 asynchronous fully non-blocking crosspoint switch ASIC [51]. Currently, they are connected to the servers using standard Ethernet cables, hence we need transceivers to convert the signal to and from the ASIC to 1000Base-T which is supported by the servers. This has significant cost and power overhead implications and is due to using standard servers instead of SoCs in the prototype platform. Each crosspoint ASIC is managed by an ARM Cortex-M3 micro-controller that configures it via an SPI serial bus and transceivers through I2C. The current design does not support 10 Gbps links, but we are in the process of designing a version with 160 10 Gbps ports using the Macom M21605 crosspoint ASIC [12]. In the experiments we use USB 1.1 to communicate with the control plane due to lack of spare Ethernet ports per server. Ethernet is supported by our switches and improves control plane latency by about an order of magnitude compared to USB.

The packet switch emulation is done in software, which allows us to understand the full functionality required before implementing it in hardware. The emulator uses two kernel drivers and a user-level process, and implements an OpenFlow-like API that provides access to the forwarding table, and callbacks on certain conditions. It binds to the six NIC ports used for internal traffic and the port used for the uplink. It also provides a virtualized NIC, to which an unmodified TCP/IP stack is bound to allow unmodified applications to be run on the testbed.

6 Evaluation

In this section we evaluate XFabric. First, we compare the performance of a reconfigurable fabric to static physical topologies. Then, we evaluate the efficiency of the algorithms used in XFabric. Finally, we show the benefits and overheads of XFabric dynamic reconfiguration.

In the experiments we evaluate XFabric using the prototype described in Section 5. In order to allow us to evaluate XFabric at scale we also use a simple flow-based simulator. We start by describing the fabric topologies, workloads and metrics used during the evaluation.

6.1 Topologies

We compare against three different topologies, two static and one dynamic. The first static topology is a 3D Torus (*3DTorus*). This topology has been widely used in HPC [15, 48] and has been proposed in data centers [2], in particular at rack scale [44]. It has the highest path length and lowest bisection bandwidth, but has a high disjoint path diversity and low cabling complexity.

The second one is a static random topology (*Random*). Random topologies have also been proposed for use in data centers [45] and are known to be "expander" graphs with high bisection bandwidth and low diameter.

We also use a dynamic reconfigurable network (*OSA*) inspired by OSA. In this network the topology is configured using the topology generation algorithm proposed in [13]. Our goal is to compare against the topology generation algorithm and we do not simulate additional features, such as the flexible link capacity described in [13]. Our implementation of the algorithm uses the same graph library as OSA [33]. This network uses a Clos circuit switch to which all the internal ports of all the SoCs are connected, so it is fully reconfigurable.

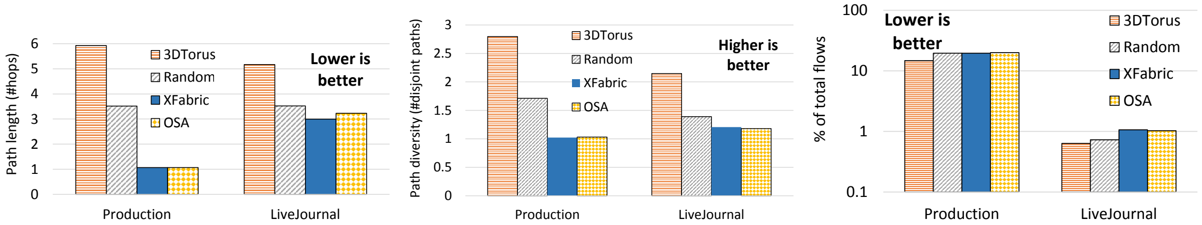
For the simulation results, unless otherwise stated, we assume the rack contains 343 SoCs each with $d = 6$ internal ports per SoC and one uplink port. In all cases we assume that the number of ports for the internal crosspoints is $s = n$ and for the uplink crosspoint $s = n + l$. We assume there are six internal crosspoints (as $d = 6$), and one uplink crosspoint with $l = 8$. Unless otherwise stated the uplinks are uniformly distributed across the SoCs and each SoC sends ingress/egress traffic to its nearest uplink in terms of path length.

We use 343 SoCs as it allows us to compare against a 3D Torus of size $7^3 = 343$. This is a challenge for XFabric, as a crosspoint with n ports establishes $n/2$ circuits so, if n is odd, one port cannot be connected to any other port on the same crosspoint. So, we will end up with 6 unused ports across all the SoCs. In practice, XFabric uses an even number of SoCs to avoid this issue. To handle the odd n configurations we form three pairs of crosspoints and in each pair statically connect a random SoC of one crosspoint to a different SoC in the second crosspoint. This means these static circuits are never reconfigured and results in strictly worse performance compared to allowing them to be reconfigured.

6.2 Workloads

We selected two workloads with well-identified traffic patterns, both based on real-world measurements.

Production cluster workload. This is a trace of 339 servers running a production workload in a mid-sized enterprise data center [7]. The data was collected over a period of 6.8 days and contains per-TCP-flow information including the source and destination IP address, the number of bytes transferred and a mapping from the servers' hostnames to the IP addresses of their NICs. We group flows based on source and destination hostnames. Flows in which the source or destination IP address does not correspond to a known hostname are considered as uplink traffic. The traffic is clustered, with heavy communication between servers with common hostname prefixes, and many-to-one traffic patterns: servers with a common hostname prefix often exchange traffic with a specific server with a different hostname prefix. This is consistent with the patterns described in [30].



(a) Path length (b) Average path diversity (c) Bottleneck link load

Figure 2: Performance summary of different fabrics for Production and LiveJournal workloads.

LiveJournal. Distributed platforms such as Pregel [34] or Tao [9] enable efficient processing of large graphs by partitioning the graph across a set of SoCs such that each SoC is assigned a set of vertices from the original graph. To generate a graph processing workload we use a trace collected from LiveJournal in December 2006 [36]. It includes 95.4% of users at that time, representing 5.2 million nodes and 48.7 million edges with an average edge degree of 18.7 edges per node. We shard the vertices into partitions using METIS, an offline graph partitioning algorithm [31] to uniformly partition the graph while minimizing traffic between partitions. There is one partition per SoC and we assume that the computation makes progress by message passing along the edges of the partitioned graph. The traffic is proportional to the number of the social graph edges between SoCs and is modeled as a constant bit rate over time.

For both workloads we map the workload onto the SoCs randomly. We also explored topology-aware workload placement using heuristic-based approaches for the static topologies [10]. For these we found that topology-aware placement always performs better than the random placement, but was always worse than the topology generated by XFabric. In practice, topology-aware placement is not easily feasible, and would often require migrating data between servers and is challenging to achieve dynamically. Due to lack of space, we only present results for random placement.

In the simulations each workload is mapped into a single traffic matrix *tm* such that for each pair of SoCs it stores the number of bytes sent and received between these SoCs. For the production trace, to scale beyond 339 SoCs, we augment the original trace by duplicating a random set of SoCs with non-zero traffic. For experiments with less than 339 SoCs, we subsample the trace by taking a random subset of the SoCs that have traffic.

6.3 Metrics

Across the experiments we use a number of metrics:

Path length. For each packet, we measure the path length from source to destination in number of hops. Since each hop adds a delay while forwarding traffic, this metric is a proxy for end-to-end latency.

Fabric	# ports	Cost	Power draw
Clos Circuit Switch	10,290	$30.9K	2.9 kW
XFabric	2,058	$6.2K	0.6 kW

Table 1: Estimated cost & power, 343 SoCs, 6 ports/SoC.

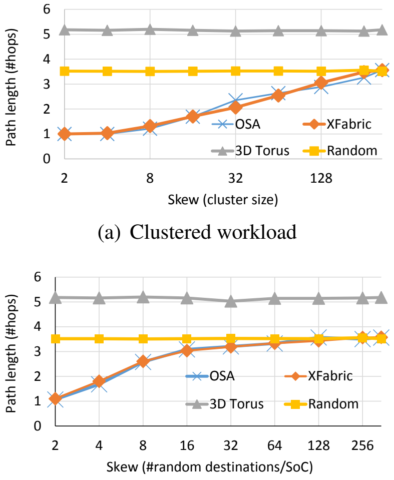
Path diversity. This metric accounts for the fault tolerance of the topology, it measures the number of disjoint shortest paths that exist for each packet. Two paths are disjoint if they share no common link. Therefore, if there are k shortest paths for all the flows in the topology, $k-1$ links can fail without impacting the average path length of the traffic. Route diversity also improves traffic load balancing allowing traffic to be spread across disjoint shortest paths.

Bottleneck link load. The metric measures the congestion within the topology by measuring the link load on the most congested link in the topology.

6.4 XFabric Performance

The two static topologies, 3D Torus and Random, do not require any additional hardware other than the switching functionality provided in the SoCs. Both the OSA and XFabric require additional ASICs to enable the reconfigurability. Any benefit obtained from being reconfigurable needs to be offset against the increased overheads this induces. Table 1 shows the number of ports required and the estimated cost and power consumption for XFabric and OSA assuming $3 per port and 0.28 W per port for 343 SoCs. OSA is a fully reconfigurable fabric supporting 2,058 ports, thus using a folded Clos. XFabric requires d crosspoint ASICs with n ports, connecting each SoC to each of the d crosspoints. This has a significant benefit in terms of cost and power.

This first simulation experiment evaluates the relative performance of the four topologies using the two workloads. For each configurable fabric we take the global demand matrix *tm* and optimize the network for *tm*. Fig-



(a) Clustered workload

(b) Random destinations workload

Figure 3: Impact of traffic skew on path length

ure 2(a) shows the average path length achieved by all the fabrics. Across both workloads the reconfigurable fabrics, XFabric and OSA, achieve shorter average path lengths than the static topologies. For Production, XFabric has an average path length of only 1.06 hops, which is 6 times less than 3D Torus and 3.7 times less than Random. For the LiveJournal workload, the path length for the reconfigurable fabrics is also lower but not by such a margin. As we will demonstrate later, the reason is due to the traffic skew. The LiveJournal workload has a lower skew. Comparing the performance of OSA and XFabric, we see for the Production workload that they both provide comparable performance. Notably for LiveJournal, even though OSA has a fully reconfigurable fabric it performs worse than XFabric. Having more flexibility in the fabric is insufficient, you also need an algorithm that can reconfigure the fabric to exploit the full flexibility.

Figure 2(b) shows the path diversity for all four fabrics with both workloads. These results show that the reconfigurable fabrics instantiate topologies with lower path diversity. The path length reduction benefits being shown in Figure 2(a) are achieved at the expense of reducing path diversity, shorter path lengths offer less opportunity for forwarding through different links. The 3D Torus has the highest path diversity, but also has the highest path length. This is an interesting trade-off where reconfigurability can provide benefit. Lowering path diversity can impact resilience to failure, and it also lowers the *aggregate* bandwidth available on the shortest paths between two SoCs. For reconfigurable fabrics, a link or SoC failure can be overcome by calculating a new topology that minimizes the impact of the failure. XFabric also can link multiple ports between the same SoCs, so providing multiple 1-hop links between two SoCs, and hence increase the aggregate bandwidth.

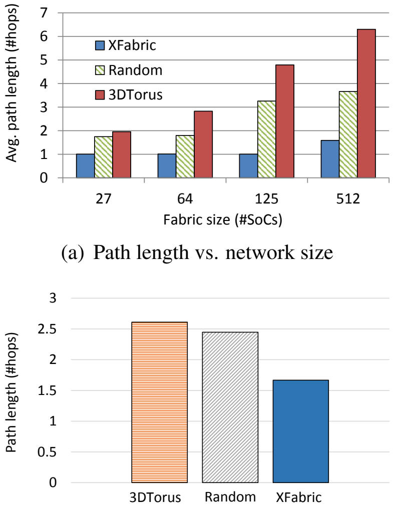
To understand this further, Figure 2(c) shows the number of flows that traverse each link. A flow from a to b is routed over the set of shortest paths in the topology between a and b and is registered on each link in the path. To achieve this each flow is split into f subflows of constant size, where f is much larger than the number of paths. The simulator estimates path congestion by counting the number of flows registered on the most loaded link in a path. To place a subflow, the simulator's transport layer checks if multiple shortest paths exist. If so two are randomly selected, and the simulator places the flow on the least congested one. This simulates traffic routing though multiple paths for any workload on top of any topology. Furthermore, the flow routing scheme ensures a good load balance of the traffic across links [37]. Figure 2(c) shows the percentage of flows that traverse the bottleneck link for each workload and topology. The most congested links on all topologies for both workloads have approximately the same load, except for the 3D Torus that benefits from its high path diversity.

The trade-off between path length and diversity also impacts the total network load across all links. The load imbalance across links is reduced when path diversity is high: in the 3D Torus the load is better balanced across links due to load balancing across multiple paths. However, because of the higher path length, the overall total load on links in the network is higher. The other topologies have a lower average total network link load than the 3D Torus, but a higher skew. However, XFabric aggressively reduces path length without significantly increasing load skew because optimization leads to links being shared across fewer source destination pairs.

We now focus on the performance of XFabric with Production, our most realistic workload, to evaluate the performance of uplink placement. Figure 4(b) shows the path length of the ingress/egress traffic in the fabric before it reaches a gateway SoC with an attached uplink. It shows that XFabric efficiently places uplinks on SoCs with heavy ingress/egress traffic: the path length is on average reduced by 32% compared to the Random topology and 37% compared to the 3DTorus.

Finally, we vary the number of SoCs in the fabric to evaluate performance at different scales. Figure 4(a) shows the average path lengths for XFabric, Random and 3D Torus topologies when the number of SoCs is varied. In the worst case, for 512 SoCs, the average path length between SoCs in the rack is 1.6 hops only, showing that optimizing the topology at up to rack-scale is beneficial.

Reconfigurable fabrics perform better for workloads with high traffic skew. To understand this more we perform a parameter sweep across different traffic skews using two synthetic workloads. For the first, called *clustered*, we partition the SoCs into clusters and each SoC



(a) Path length vs. network size

(b) Uplink placement

Figure 4: Scalability and uplink placement performance.

communicates with all other SoCs in the same cluster. We vary the number of SoCs per cluster between 2 and 343. Intuitively, this results in a set of traffic matrices in which the traffic skew grows as cluster size drops. Figure 3(a) shows the path length as a function of the cluster size for XFabric and OSA with the clustered workload. The cluster size has no impact on static topologies because no reconfiguration is performed. When the skew is high, reconfigurable topologies are able to more efficiently optimize for the skew, up to the point when most of the traffic is sent through 1 hop. As the cluster size increases, the traffic pattern shifts to an all-to-all pattern and performance of reconfigurable fabrics becomes comparable to a Random topology. Notably, there is almost no difference between XFabric and OSA.

We run a second experiment with a different workload to evaluate the impact of the traffic pattern on path length. For this workload, called *random destinations*, each SoC sends traffic to a random set of k SoCs in the rack. For low values of k, the workload is very skewed and as it increases the workload progressively adopts an all-to-all traffic pattern. However, this results in a less clustered workload, even when traffic is very skewed. Figure 3(b) shows the path length as a function of the number of destinations per SoC for all fabrics. We observe the same trend as for the clustered workload, with both OSA and XFabric outperforming static topologies by up to a factor of 3.5 when the skew is high.

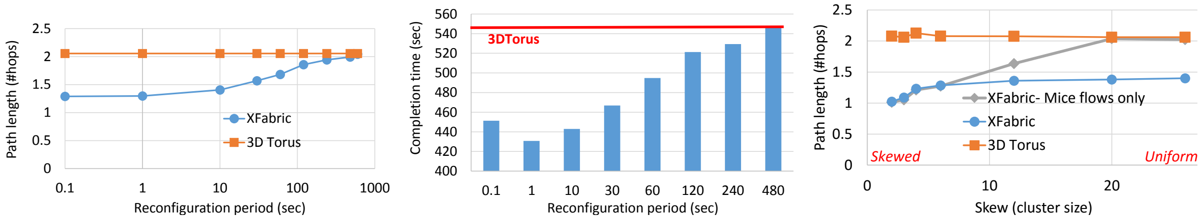
6.5 XFabric Prototype Performance

So far we evaluated the benefits of XFabric at scale using our simulator. In the next experiment we use our prototype platform to evaluate the dynamic reconfiguration performance of XFabric. Frequent XFabric reconfiguration is beneficial as it improves the responsiveness of the fabric to changes in traffic load, improving performance. However, too frequent reconfiguration induces overheads at the packet switching layer as it may result in packet loss. The reconfiguration of the crosspoints at layer 1 is not synchronized with layer 2. Too frequent packet loss can have a negative impact on the throughput at the transport layer, particularly if TCP is used.

We have created a test framework that uses unmodified TCP and replays flow-level traces derived from the Production workload. The framework opens a new socket for each flow and starts six flows per SoC concurrently, operating as a closed loop per SoC, so when one flow finishes the next is started on the SoC. In each experiment we configure the network as a 3D Torus and do not allow the network to reconfigure for the first 2 minutes. Unless otherwise stated the flow size is selected from the distribution of flow sizes in the Production workload, which is a typical heavy tailed distribution with a small number of elephant flows and a high number of mice flows, and an average flow size of 9.3 MB.

We first evaluate the impact of reconfiguration frequency on performance. We generate a trace with 250,000 flows and vary the reconfiguration period of XFabric between $t = 0.1$ to 480 seconds until the trace run is completed. Figure 5(a) shows the average path length of each packet as a function of the reconfiguration period. As expected, decreasing the reconfiguration period reduces the path length. When reconfigured every 30 seconds or less, XFabric achieves more than a 25% reduction in path length compared to the 3D Torus. The path length is reduced by approximately 37% (from 2.05 to 1.28 hops) for a reconfiguration interval of a second or less. This shows that even at small scale, reconfiguring the topology significantly reduces path length.

In order to understand how reconfiguring the fabric impacts goodput, Figure 5(b) shows the average completion time as a function of the reconfiguration interval. For each run we define completion time as the execution time from 2 minutes (when reconfiguration is enabled) to the end of the trace. The completion time for the 3D Torus is denoted by the red line and is constant as it does not reconfigure. We can see that for XFabric, shorter path length also reduces completion time, because each flow uses less network resources, increasing overall goodput. The completion time is reduced by 20% compared to the 3D Torus for a reconfiguration interval of 1 second. When reconfigured every 100 ms, the completion time increases compared to the 1 second interval, despite the path length being similar for both intervals. This shows the trade-off between the benefit of reconfiguration versus potential impact of packet loss on the transport layer.



(a) Path length (b) Completion time (c) Impact of workload skew on path length

Figure 5: Prototype performance.

We now explore the impact of traffic skew on performance. We set the reconfiguration period to 1 second and generate a set of traces in which SoCs are divided into clusters of fixed size c, with $c = 2$ to $c = 27$. Each trace has 250,000 flows and for each SoC, the destination of each flow is randomly selected in the corresponding cluster. Hence for $c = 27$ the traffic is uniform, and traffic skew increases as cluster size drops. Figure 5(c) shows the average path length as a function of the cluster size. As expected, for XFabric the path length is lower when the skew is high. In the extreme, when $c = 2$, the average path length is 1.02, which is more than a factor of 2 better compared to the 3D Torus. Notably, XFabric still has a 35% lower path length than the 3D Torus when the traffic is uniform. This is because many elephant flows live long enough to benefit from reconfiguration.

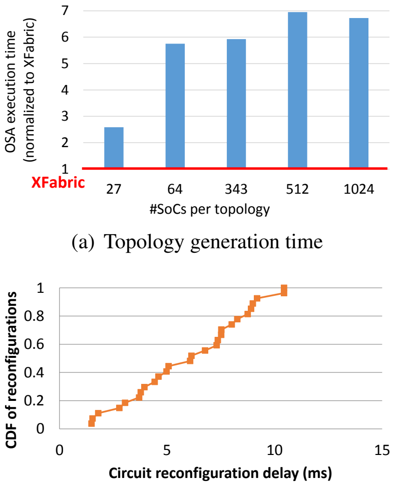
To quantify the impact of elephant flows, we generate a set of traces in which all flows are smaller than the median value from the Production flow size distribution. Each trace has 17 million flows with an average flow size of 129 KB and a maximum of 365 KB. SoCs are divided into clusters as previously but each SoC sends sequences of ten short flows to destinations in its cluster. Each SoC thus has a relatively stable flow rate, but per-destination traffic is bursty, which can be compared to real traffic patterns [43]. Figure 5(c) shows that when the traffic is skewed, XFabric is still able to accurately estimate the demand without elephant flows because each SoC has a limited number of destinations and the traffic pattern is predictable. However, as the traffic gets uniform, XFabric progressively loses the ability to accurately estimate the demand and the path length becomes comparable to the static topology.

6.6 Reconfiguration Overheads

We now look at the overheads associated with XFabric reconfiguration. In the first experiment we calculate the average execution time across five runs for OSA and XFabric to generate a new topology for topology sizes ranging from 27 to 1024 SoCs. Figure 6(a) shows the time taken for OSA normalized to XFabric. In all cases XFabric significantly outperforms OSA. For 512 SoCs and below, XFabric generates topologies in less than 700ms, while for 1024 SoCs, the topology is generated in about 3 seconds. This shows that XFabric is able to optimize any rack-scale topology fast enough for dynamic reconfiguration in seconds or less. In comparison, it takes OSA about 20.5 seconds for the largest topologies, which is over 6 times longer.

The next experiment measures the end-to-end reconfiguration latency of XFabric. At the beginning, XFabric is configured as a 3D Torus and runs an all-to-all workload for 60 seconds to allow it to reach steady state. The controller then generates a new topology and reconfigures the data plane. Figure 6(b) shows the CDF of reconfiguration delays; each data point is the time taken for each server to have pushed a new forwarding table into the local packet switch from when the first crosspoint ASIC was reconfigured. Figure 6(b) shows that all servers are reconfigured within 11 ms. The latency for each micro-controller attached to the crosspoint ASIC to internally reconfigure it is approximately 40 microseconds. This delay is currently dominated by two factors, first the latency of the control plane interface which uses USB 1.1 and has a 1 ms delay, combined with the fact that the current prototype controller sequentially communicates with each of the micro-controllers, hence the last crosspoint ASIC is reconfigured 8 ms after the first. This latency would be removed if the controller used an Ethernet-based control plane.

We now measure the packet loss rate due to reconfiguration in the data plane. We run 5 experiments with the first workload described in Section 6.5 and a reconfiguration period set to 1 second. We count all Ethernet frames sent and received through each NIC on each SoC. Ethernet frames that are corrupted due to reconfiguration fail the CRC check and are dropped on the receiver before being counted. Hence the difference between the total number of frames sent and received by all SoCs accounts for the loss. We conservatively assume that all lost frames are due to reconfiguration. The average loss rate is 0.69 frames per full-duplex link per reconfiguration. The crosspoint ASICs we use have a switching time of



(a) Topology generation time

(b) Reconfiguration time

Figure 6: Reconfiguration overheads.

20 ns [51], which is the time to transmit 3 bytes at 1 Gbps and 25 bytes at 10 Gbps. With a minimal Ethernet frame size of 64 bytes [28], we expect the worst case loss on a full-duplex link to be 4 frames per reconfiguration for both 1 and 10 Gbps.

7 Related Work

The XFabric design is heavily influenced by the Calxeda SoC design, the first publicly available SoC that incorporates a packet switch. This SoC also explicitly provisioned ports for internal communication and a single Ethernet uplink port per SoC and we assumed this model for XFabric. We believe that this is a likely design point for other SoCs. Calxeda unfortunately collapsed, but we believe that other chip vendors will likely move in this direction. For example, the Xeon-D processor designed by Intel [52] is a low-power SoC with two 10 Gbps ports per SoC. Oracle recently announced their next-generation SPARC design with two 56 Gbps Infiniband controllers co-located with the CPU on silicon [26]. However, currently none of these designs yet supports embedded packet switches.

Optical Circuit Switching (OCS) has been proposed to establish physical circuits between ToR switches at the data center scale [20, 47]. They rely on MEMS-based switches and have high reconfiguration latency. To address latency sensitive traffic, c-Through and Helios rely on a separate packet switched network. Mordia [41] routes latency sensitive traffic through circuits by time-sharing the circuits between servers in a rack. XFabric differs from these architectures because they do not route packets over multiple circuits when a direct circuit is not available. The closest to our proposal is OSA [13] that allows multi-hop data forwarding between ToRs (servers in a rack use traditional packet switching). However, OSA does not address the issue of scaling beyond a single circuit switch and assumes all ToRs have all reconfigurable ports connected to the same switch. Compared to OSA, XFabric addresses a set of challenges unique to the rack scale. It reduces the space and power consumed by the reconfigurable fabric by using several smaller cross-point ASICs and deals with uplink management.

Halperin *et al.* [23] propose to augment standard data center networks with wireless flyways used to decongest traffic hotspots. Zhou *et al.* [49] improve the technique by bouncing the signal off the data center ceiling to overcome physical obstacles. However, the wireless technology has a set of physical constraints (e.g. signal interference) due to which only a subset of the links are reconfigurable while the rest of the traffic is still routed through the traditional network. FireFly [24] is a data center level architecture in which the physical layer is supported by lasers reflected using large ceiling mirrors. However, this technique is hard to leverage inside a rack.

In HPC, Kamil et al. uses an optical circuit switch to interconnect packet switches, which can then be pooled to increase available bandwidth between heavily communicating servers [29]. This work differs from XFabric as it considers one large circuit switch for all packet switches and leverages the high predictability of HPC workloads to compute efficient topologies.

AN3 [19] performs virtual circuit switching and allows speculative circuit establishment supported by custom switches implemented in FPGA. This system differs from our work as it establishes circuits at layer 2 of the network and operates over a static physical topology.

8 Conclusion

Emerging hardware trends and server densities are going to challenge the usual approach of connecting all the servers in a rack to a single ToR switch. One explored solution is to disaggregate the packet-switching functionality across SoCs. Based on the observation that different network topologies support different workloads we propose XFabric, a dynamically reconfigurable rack-scale fabric. It differs from prior work by addressing specific requirements that arise at rack scale, dealing with power and space constraints and managing uplink to the data center network. A prototype XFabric implementation demonstrates the reconfiguration benefits and shows that partial reconfigurability achieves the performance of full reconfigurability at lower cost and power consumption.

Acknowledgments

We are grateful to the anonymous reviewers, and in particular to our shepherd S. Keshav, for their feedback.

References

[1] HP Moonshot System: The World's First Software-Defined Server -Family guide, Jan. 2014.

[2] ABU-LIBDEH, H., COSTA, P., ROWSTRON, A., O'SHEA, G., AND DONNELLY, A. Symbiotic Routing in Future Data Centers. *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 51–62.

[3] AFCOM. Data center Standards. `http://bit.ly/1KPoZOZ`.

[4] Amazon joins other web giants trying to design its own chips. `http://bit.ly/1J5t0fE`.

[5] ASANOVIC, K., AND PATTERSON, D. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *USENIX FAST* (2014).

[6] BALAKRISHNAN, S., BLACK, R., DONNELLY, A., ENGLAND, P., GLASS, A., HARPER, D., LEGTCHENKO, S., OGUS, A., PETERSON, E., AND ROWSTRON, A. Pelican: A Building Block for Exascale Cold Data Storage. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (2014), USENIX Association, pp. 351–365.

[7] BALLANI, H., JANG, K., KARAGIANNIS, T., KIM, C., GUNAWARDENA, D., AND O'SHEA, G. Chatty Tenants and the Cloud Network Sharing Problem. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation* (2013), USENIX Association, pp. 171–184.

[8] BOSTON. Boston Viridis Data Sheet. `http://download.boston.co.uk/downloads/9/3/2/932c4ecb-692a-47a9-937d-a94bd0f3df1b/viridis.pdf`.

[9] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., ET AL. Tao: Facebooks Distributed Data Store for the Social Graph. In *USENIX ATC* (2013).

[10] BURKARD, R., PARDALOS, P., AND PITSOULIS, L. The Quadratic Assignment Problem. In *Handbook of Combinatorial Optimization* (1998), Kluwer Academic Publishers, pp. 241–338.

[11] Calient S320 Optical Circuit Switch Datasheet. `http://www.calient.net/download/s320-optical-circuit-switch-datasheet/`.

[12] Macom M21605 Crosspoint Switch Specification. `http://www.macom.com/products/product-detail/M21605/`.

[13] CHEN, K., SINGLA, A., SINGH, A., RAMACHANDRAN, K., XU, L., ZHANG, Y., WEN, X., AND CHEN, Y. OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility. *IEEE/ACM Transactions on Networking (TON) 22*, 2 (2014), 498–511.

[14] COSTA, P., BALLANI, H., RAZAVI, K., AND KASH, I. R2C2: A Network Stack for Rack-Scale Computers. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), ACM, pp. 551–564.

[15] CRAY. CRAY XT3 Datasheet. `http://www.craysupercomputers.com/downloads/CrayXT3/CrayXT3_Datasheet.pdf`.

[16] DAGLIS, A., NOVAKOVIC, S., BUGNION, E., FALSAFI, B., AND GROT, B. Manycore Network Interfaces for In-Memory Rack-Scale Computing. In *Proceecidings of the 42nd International Symposium in Computer Architecture* (2015), no. EPFL-CONF-207612.

[17] Dell PowerEdge c5220 Microserver. `http://www.dell.com/us/business/p/poweredge-c5220/pd`.

[18] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FARM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI* (2014), vol. 14.

[19] ERIC CHUNG, ANDREAS NOWATZYK, TOM RODEHEFFER, CHUCK THACKER, AND FANG YU. AN3: A Low-Cost, Circuit-Switched Datacenter Network. Tech. Rep. MSR-TR-2014-35, March 2014.

[20] FARRINGTON, N., PORTER, G., RADHAKRISHNAN, S., BAZZAZ, H. H., SUBRAMANYA, V., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. Helios: a Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 339–350.

[21] Intel, Facebook Collaborate on Future Data Center Rack Technologies. `http://intel.ly/MRpOMO`.

[22] Google Ramps Up Chip Design. `http://ubm.io/1iQooNe`.

[23] HALPERIN, D., KANDULA, S., PADHYE, J., BAHL, P., AND WETHERALL, D. Augmenting Data Center Networks with Multi-Gigabit Wireless Links. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 38–49.

[24] HAMEDAZIMI, N., QAZI, Z., GUPTA, H., SEKAR, V., DAS, S. R., LONGTIN, J. P., SHAH, H., AND TANWER, A. FireFly: a Reconfigurable Wireless Data Center Fabric using Free-Space Optics. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2014), ACM, pp. 319–330.

[25] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving High Utilization with Software-Driven WAN. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 15–26.

[26] Oracles Sonoma Processor. `http://www.hotchips.org/archives/2010s/hc27/`.

[27] HP ProLiant m800 Server Cartridge. `http://bit.ly/1JxM9Zr`.

[28] IEEE. 802.3-2012 IEEE Standard for Ethernet. `http://standards.ieee.org/findstds/standard/802.3-2012.html`.

[29] KAMIL, S., PINAR, A., GUNTER, D., LIJEWSKI, M., OLIKER, L., AND SHALF, J. Reconfigurable Hybrid Interconnection for Static and Dynamic Scientific Applications. In *Proceedings of the 4th international conference on Computing frontiers* (2007), ACM, pp. 183–194.

[30] KANDULA, S., SENGUPTA, S., GREENBERG, A., PATEL, P., AND CHAIKEN, R. The Nature of Data Center Traffic: Measurements & Analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference* (2009), ACM, pp. 202–208.

[31] KARYPIS, G., AND KUMAR, V. Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *Supercomputing* (1998).

[32] KRUSKAL, J. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical society 7*, 1 (1956), 48–50.

[33] LEMON Graph Library. `http://lemon.cs.elte.hu/trac/lemon`.

[34] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a System for Large-Scale Graph Processing. In *SIGMOD* (2010).

[35] Microservers Powered by Intel. `http://www.intel.com/content/www/us/en/servers/microservers.html`.

[36] MISLOVE, A., MARCON, M., GUMMADI, K. P., DRUSCHEL, P., AND BHATTACHARJEE, B. Measurement and Analysis of Online Social Networks. In *IMC* (2007).

[37] MITZENMACHER, M. The Power of Two Choices in Randomized Load Balancing. *Parallel and Distributed Systems, IEEE Transactions on 12*, 10 (2001).

[38] How Microsoft Designs its Cloud-Scale Servers. `http://bit.ly/1HKCy27`.

[39] NOVAKOVIC, S., DAGLIS, A., BUGNION, E., FALSAFI, B., AND GROT, B. Scale-Out NUMA. *ACM SIGARCH Computer Architecture News 42*, 1 (2014), 3–18.

[40] OpenFlow Specification. `http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf`.

[41] PORTER, G., STRONG, R. D., FARRINGTON, N., FORENCICH, A., SUN, P., ROSING, T., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. Integrating Microsecond Circuit Switching into the Data Center. In *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013* (2013), D. M. Chiu, J. Wang, P. Barford, and S. Seshan, Eds., ACM, pp. 447–458.

[42] PUTNAM, A., CAULFIELD, A. M., CHUNG, E. S., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., GOPAL, G. P., GRAY, J., ET AL. A Reconfigurable Fabric for Accelerating Large-Scale Data Center Services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on* (2014), IEEE, pp. 13–24.

[43] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), ACM, pp. 123–137.

[44] SEAMICRO, A. AMD SeaMicro SM15000 Fabric Compute Systems. `http://www.seamicro.com/sm15000`.

[45] SINGLA, A., HONG, C.-Y., POPA, L., AND GODFREY, P. B. Jellyfish: Networking Data Centers Randomly. In *NSDI* (2012), vol. 12, pp. 17–17.

[46] SUDAN, K., BALAKRISHNAN, S., LIE, S., XU, M., MALLICK, D., LAUTERBACH, G., AND BALASUBRAMONIAN, R. A Novel System Architecture for Web-Scale Applications using Lightweight CPUs and Virtualized I/O. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on* (2013), IEEE, pp. 167–178.

[47] WANG, G., ANDERSEN, D. G., KAMINSKY, M., PAPAGIANNAKI, K., NG, T., KOZUCH, M., AND RYAN, M. c-Through: Part-Time Optics in Data Centers. *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 327–338.

[48] WWW.HPCRESEARCH.NL. IBM BlueGene P&Q. `http://www.hpcresearch.nl/euroben/Overview/web12/bluegene.php`.

[49] ZHOU, X., ZHANG, Z., ZHU, Y., LI, Y., KUMAR, S., VAHDAT, A., ZHAO, B. Y., AND ZHENG, H. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 443–454.

[50] Analog Devices ADN4612. `http://www.analog.com/media/en/technical-documentation/data-sheets/ADN4612.pdf`.

[51] Analog Devices ADN4605. `http://www.analog.com/en/products/switches-multiplexers/digital-crosspoint-switches/adn4605.html`.

[52] Intel Xeon Processor D-1500 Family Product Brief. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-d-brief.html`.

[53] 10GBase-KR FEC Tutorial. `http://www.ieee802.org/802_tutorials/06-July/10GBASE-KR_FEC_Tutorial_1407.pdf`.

[54] Vitesse VSC3144. `https://www.vitesse.com/products/product/VSC3144`.

Be Fast, Cheap and in Control with SwitchKV

Xiaozhou Li[1], Raghav Sethi[1], Michael Kaminsky[3], David G. Andersen[2], Michael J. Freedman[1]

[1]*Princeton University,* [2]*Carnegie Mellon University,* [3]*Intel Labs*

Abstract

SwitchKV is a new key-value store system design that combines high-performance cache nodes with resource-constrained backend nodes to provide load balancing in the face of unpredictable workload skew. The cache nodes absorb the hottest queries so that no individual backend node is over-burdened. Compared with previous designs, SwitchKV exploits SDN techniques and deeply optimized switch hardware to enable efficient content-based routing. Programmable network switches keep track of cached keys and route requests to the appropriate nodes at line speed, based on keys encoded in packet headers. A new hybrid caching strategy keeps cache and switch forwarding rules updated with low overhead and ensures that system load is always well-balanced under rapidly changing workloads. Our evaluation results demonstrate that SwitchKV can achieve up to 5× throughput and 3× latency improvements over traditional system designs.

1 Introduction

In pursuit of meeting aggressive latency and throughput service level objectives (SLOs) for Internet services, providers have increasingly turned to in-memory [32, 35] or flash-based [2] key-value storage systems as caches or primary data stores. These systems can offer microseconds of latency and provide throughput hundreds to thousands of times that of the hard-disk-based approaches of yesteryear. The choice of flash vs DRAM comes with important differences in throughput, latency, persistence, and cost-per-gigabyte. Recent advances in SSD performance, including new hardware technologies such as NVMe [34], are opening up new points in the design space of storage systems that were formerly the exclusive domain of DRAM-based systems. However, no single SSD is fast enough, and scale-out designs are necessary both for capacity and throughput.

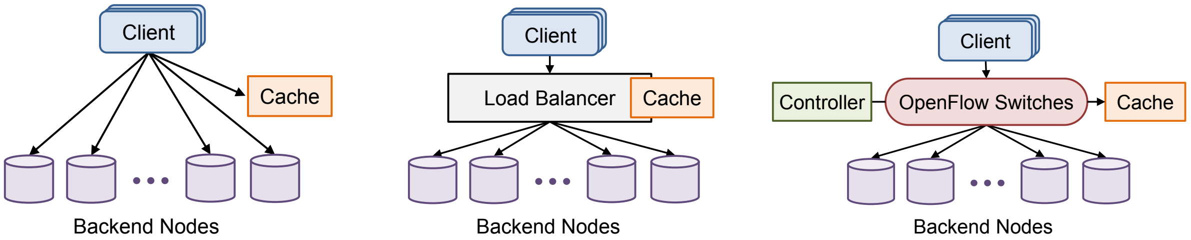
Dynamic load balancing is a key challenge to scaling out storage systems under skewed real-world workloads [3, 5, 7]. The system performance must not become bottlenecked due to unevenly partitioned load across cluster nodes. Conventional static data partitioning techniques such as consistent hashing [23] do not help with dynamic load imbalance caused by skewed and rapidly-changing key popularity. Load balancing techniques that reactively replicate or transfer hot data across nodes often introduce performance and complexity overheads [24].

Prior research shows that a small, fast frontend cache can provide effective dynamic load-balancing by directly serving the most popular items without querying the backend nodes, making the load across the backends much more uniform [13]. That work proves that the cache needs to store only the $O(n \log n)$ hottest items to guarantee good load balance, where n is the total number of *backend nodes* (independent of the number of keys).

Unfortunately, traditional caching architectures such as the look-aside Memcached [15] and an on-path look-through small cache [13] suffer a major drawback when using a frontend cache for load balancing, as shown in Table 1. In these architectures, clients must send all read requests to the cache first. This approach imposes high overhead when the hit ratio is low, which is the case when the cache is small and used primarily for load balancing. Some look-aside systems (Fig. 1a) make the clients responsible for handling cache misses [15], which further increases the system overhead and tail latency. Other designs that place the cache in the frontend load-balancers (Fig. 1b) [13] are vulnerable to the frontend crashes.

SwitchKV is a new cluster-level key-value store architecture that can achieve high efficiency under widely varying and rapidly changing workloads. As shown in Fig. 1c, SwitchKV uses a mix of server classes, where specially-configured high-performance nodes serve as fast, small caches for data that is hash partitioned across resource-constrained backend nodes. At the heart of SwitchKV's design is an efficient content-based routing mechanism that uses software-defined networking (SDN) techniques to serve requests with minimal overhead. Clients encode keys into packet headers and send the requests to OpenFlow switches. These switches maintain forwarding rules for all cached items, and route requests directly to the cache or backend nodes as appropriate based on the embedded keys.

SwitchKV achieves high performance by *moving the cache out of the data path* and by exploiting switch hardware that has already been optimized to match (on query keys) and forward traffic to the right node at line rate with



(a) Look-aside **(b)** On-path look-through **(c)** SwitchKV

Figure 1: Different cache architectures.

	Look-aside	**On-path look-through**	**SwitchKV**
Clients' responsibilities	handle cache misses	nothing (transparent)	encode keys in packet headers
Cache load	*100% queries*	*100% queries*	cache hits (likely <40% queries)
Latency with cache miss	*three* machine transits	*two* machine transits	one machine transit
Failure points	switches	*load balancer*, switches	switches
Cache update involves	cache, backends	cache, backends	cache, backends, *switches*
Cache update rate limit	high	high	*low* (<10K/s in switch hardware)

Table 1: Comparison of different cache architectures.

low latency. All responses return within one round-trip, and there is no overhead for the significant volume of queries for keys that are not in the cache. SwitchKV can scale-out by adding more cache nodes and switches, and is resilient to cache crashes.

The benefits of using OpenFlow switches come at a price: the update rate of forwarding rules in hardware is much lower than that of in-memory caches. Our solution includes an efficient hybrid cache update mechanism that minimizes the cache churn, while still reacting quickly to rapid changes in key popularity. Backends send periodic reports to the cache nodes about their recent hot keys as well as instant reports about keys that suddenly become very popular. Cache nodes maintain query statistics for the cached keys, add or evict the appropriate keys when they receive reports, and instruct SDN controllers to update switch forwarding rules accordingly.

Our SwitchKV prototype uses low-power backend nodes. The same design principles and evaluation results also apply to clusters with more powerful backends, by using high-end cache servers [26] that can keep the same order of performance gap between cache and backends.

The main contributions of this paper are as follows:

- The design of a new cost-effective, large-scale, persistent key-value store architecture that exploits SDN and switch hardware capabilities to enable efficient cache-based dynamic load balancing with content routing.
- An efficient cache update mechanism to meet the challenges imposed by switch hardware and small cache size, and to react quickly to rapid workload changes.
- Evaluation and analysis that shows SwitchKV can handle the traffic characteristics of modern cloud applications more efficiently than traditional systems.

2 Background and Related Work

Clustered Key-Value Stores. Their simple APIs (e.g., `get`, `put`, `delete`) form a fundamental building block of modern cloud services. Given the performance requirements, some systems keep data entirely in memory [35, 38], with disks used only for failure recovery; others put a significant fraction of data in cache to achieve high hit ratio [32]. Systems that aggressively use DRAM are often more expensive and power-hungry than those that use flash storage.

Meanwhile, SSDs are becoming faster, as hardware [34] and software stacks [27, 39] for flash storage become optimized. With a proper design, SSD-based key-value store clusters can be a cost-effective way to meet the SLOs of many cloud services.

Load Balancing. Key-value workloads for cloud applications are often skewed [3, 7]. Many cloud services further experience unpredictable flash crowds or adversarial access patterns [22]. These all pose challenges to scaling out SSD-based key-value clusters, because the service quality is often bottlenecked by the performance of overloaded nodes. For example, a web server may need to contact 10s to 100s of storage nodes with many sequential requests when responding to a page request [32], and the tail latency can significantly degrade the service performance at large scale [10].

Good load balancing is necessary to ensure that the cluster can meet its performance goals without substantial over-provisioning. Consistent hashing [23] and virtual nodes [9] help balance the static load and space utilization, but are unable to balance the dynamic load with skewed query distributions. Traditional dynamic load balancing methods either use the "power of two choices" [31] or migrate data between nodes [6, 24, 40]. Both are limited in their ability to deal with large skew, are usually too slow to handle rapid workload changes, and often introduce consistency challenges and system overhead for migration or replication.

Caching can be an effective dynamic load balancing technique for hash partitioned key-value clusters [4, 13]. A frontend cache can absorb the hottest queries and make the load across the backends less skewed. Fan et al. [13] prove that the size of the cache required to provide good load balance is only $O(n \log n)$, where n is the total number of backend nodes. This theoretical result inspired the design of SwitchKV.

Caching Architectures. Look-aside [15] and on-path look-through [13] are the two typical caching architectures, shown in Fig. 1 and compared in Table 1. When the cache is small, the hit ratio is usually low (e.g., <40%). This is enough to ensure good load balance, but creates serious overhead in both traditional architectures. The cache is required to process all queries, including those for keys that are not cached, wasting substantial system I/O and network bandwidth in the process.

A cache miss in a look-aside architecture results in an additional round-trip of latency, as the query must be sent back to the client with a cache miss notification, and then resent to the backend. Look-through architectures reduce this latency by placing the cache in the on-path load balancer, however, the cache still must process each incoming request to determine whether to forward or serve it. Additionally, the load balancers become new critical failure points, which are far less reliable and durable than network switches [16].

3 SwitchKV Design

The primary design goal of SwitchKV is to remove redundant components on the query path such that latency can be minimized for all queries, throughput can scale out with the number of backend nodes, and availability is not affected by cache node failures.

The key to achieving this goal is the observation that specialized programmable network switches can play a key role in the caching system. Switch hardware has been optimized for decades to perform basic lookups at high speed and low cost. This simple but efficient function is a perfect match to the first step of a query processing: determine whether the key is cached or not.

The core of our new architectural design is an effective content-based routing mechanism. All clients, cache nodes, and backend nodes are connected with OpenFlow switches, as shown in Fig. 1c. Clients encode keys in query packet headers, and send packets to the cluster switch. Switches have forwarding rules installed, including exact match rules for each cached key and wildcard rules for each backend, to route queries to the right node at line rate. Table 1 summarizes the significant benefits of this new architecture over traditional ones.

Exploiting SDN and fast switch hardware benefits system performance, efficiency and robustness. However, it also adds complexity and limitations. The switches have limited rule space and a relatively slow rule update rate. Therefore, cached keys and switch forwarding rules must be managed carefully to realize the full benefits of this new architecture. The rest of this section describes SwitchKV's query-processing flow and mechanisms to keep the cache up-to-date.

3.1 Content Routing for Queries

We first describe how SwitchKV handles client queries, assuming both cache and switch forwarding rules are installed and up-to-date. The process of updating cache and switch rules will be discussed in Section 3.2.

Query operations are performed over UDP, which has been widely used in large-scale, high-performance in-memory key-value systems for low latency and low overhead [28, 32]. Because UDP is connectionless, queries can be directed to different servers by switches without worrying about connection states. With a well-provisioned network, packet loss is rare [32], and simple application-driven loss recovery is sufficient to ensure both reliability and high throughput [28].

3.1.1 Key Encoding and Switch Forwarding

An essential system component to enable content-based routing is the programmable network switches that can install new *per cached key* forwarding rules on the fly. These switches can use both TCAM and L2/L3 tables for packet processing. The TCAM is able to perform flexible wildcard matches, but it is expensive and power hungry to include on switches. Thus, the size of the TCAM table is usually limited to a few thousand entries [25, 37].[1] The L2 table, however, matches only on destination MAC addresses; it can be more cost-effectively implemented in switches and is more power-efficient. Modern commodity switches support 128K [37] or more L2 entries. These sizes may be insufficient for environments where a large percentage of data must be cached, but is a large enough cache size to ensure good load balancing in SwitchKV.

Key Encoding in Packet Headers. Because MAC addresses have more bits for key encoding and switches usually have large enough L2 tables to store forwarding rules for all cached keys, clients encode query keys in the destination MAC addresses of UDP packets. The MAC

[1] Some high-end switches advertise larger TCAM table (e.g., 125K to 1 million entries [33]), albeit at higher cost and power consumption. Such capabilities would not meaningfully change our design, as our design primarily relies on exact-match rules.

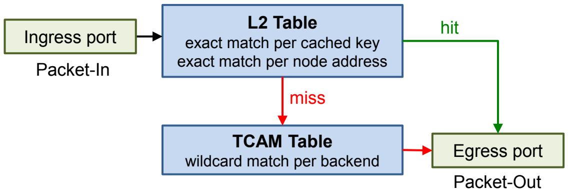

Ingress port
Packet-In
L2 Table
exact match per cached key
exact match per node address
hit
miss
TCAM Table
wildcard match per backend
Egress port
Packet-Out

Figure 2: Packet flow through a switch.

consists of a small *prefix* and a *hash* of the key, computed by the same consistent hashing used to partition the keyspace across the backends.

The prefix is used to identify the packet as being a request destined for SwitchKV, and to let the switches distinguish different types of queries. Only `get` queries coming directly from the clients may need to be forwarded to the cache nodes. Other types of queries should be forwarded to the backends, including `put` queries, `delete` queries, and `get` queries from a cache node due to cache misses. Therefore, get queries from the clients use one prefix, and all other queries use a different one.

In order to forward queries to the appropriate backend nodes, each client tracks the mapping between keyspace partitions and the backend nodes, and encodes *identifiers of backends* for the query keys into the destination IP addresses. This mapping changes only when backend nodes are added or removed, so client state synchronization has very low overhead.

Finally, the client's address and identity information is stored in the packet payload so that the node that serves the request knows where to send responses.

Switch Forwarding. There are three classes of rules in switches, which are used to forward `get` queries for the cached keys to the cache nodes, other queries to the backends, and non-query packets (e.g., query responses, cache updates) to the destination node respectively. Fig. 2 shows the packet flow through a switch. The L2 table stores exact match rules on destination MAC addresses for each cached key and each cache and backend node. The TCAM table stores wildcard match rules on destination IP addresses for each backend node.

The L2 table is set to have a higher priority. A switch will first look for an exact match in the L2 table and will forward the packet to an egress port if either the packet was addressed directly to a node or it is a `get` query for a cached key. If there is no match in the L2 table, the switch will then look for a wildcard match in the TCAM and forward the packet to the appropriate backend node.

Below are the detailed switch forwarding rules:

- Exact match rules in L2 table for all cached keys. We use `pre1` to denote the prefix for `get` queries from clients. For each cached key in cache node:
  ```
  match:<mac_dst = pre1-keyhash>
  action:<port_out = port_cache_node>
  ```

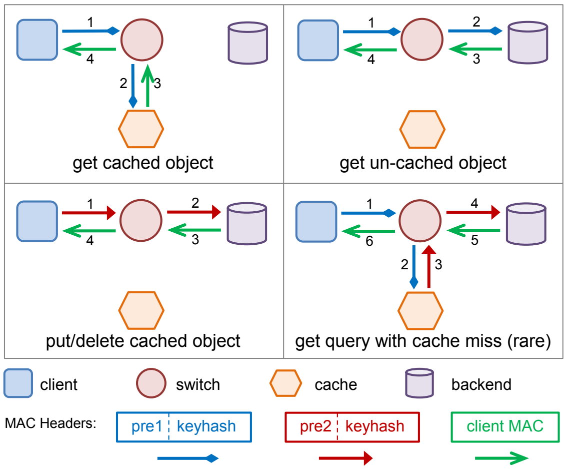


Figure 3: Query packets flows and destination MAC addresses. Internal messages for cache consistency during `put` or `delete` operations are not included. A cache miss only occurs due to key hash collision or temporarily outdated switch rules.

- Exact match rules in L2 table for all clients, cache nodes, and backends. For each node:
  ```
  match:<mac_dst = mac_node>
  action:<port_out = port_node>
  ```
- Wildcard match rules in TCAM table for all backend nodes. For each backend node:
  ```
  match:<ip_dst/mask = id_node>
  action:<port_out = port_backend_node>
  ```

3.1.2 Query Flow Through the System

A main benefit of SwitchKV is that it can send queries to the appropriate nodes with minimal overhead, especially for queries on uncached keys which make up most of the traffic. Fig. 3 shows the possible packet flows of queries.

Handle Read Requests. SwitchKV targets read-heavy workloads, so the efficiency of handling read requests is critical to the system performance. Switches route `get` queries to the cache or backends based on match results in the forwarding tables. When it receives a `get` query, the cache or backend node will look for the key in its local store, either in memory or SSD. The backend will send a reply message with the destination MAC set as the client address. The cache node will also reply if the key is found. This reply will be forwarded back to the client.

In most cases, queries sent to the cache node will hit the cache, because queries for keys not in the cache were filtered out by the switches. However, it is possible for a cache node to receive a `get` query but not find the key in its local in-memory store. This may occur due to a small delay in rule removal from the switch, or a rare hash collision with another key. When this happens, the cache node must forward the packet to the backends. To do so, the cache will send the query packet back to the switch, with the appropriate destination MAC address prefix (e.g., from `pre1` to `pre2` in Fig. 3). This prevents the packet from matching the same L2 rule in switches again, so that the query can be forwarded to the appropriate backend node via a wildcard match in TCAM.

Handle Write Requests. Clients send `put` and `delete` queries with a MAC prefix that is different from the prefix of `get` queries (as shown in Fig. 3), so that the packets will not trigger a rule in the L2 table of switches, and will be forwarded directly to the backends. When a backend node receives a `put` or `delete` query for a key, it will update its local data store and reply to the client.

Each backend node keeps track of which keys in its local store are also being cached. If a `put` or `delete` request for a cached key arrives, the backend will send messages to update the cache node before replying to the clients. The cache node is then responsible for communicating the update to the network controller for switch rule updates. This policy ensures that data items in the cache and backends are consistent to the client, but allows temporary inconsistency between cached keys and switch forwarding rules. The next section describes the detailed mechanism of cache update and consistency.

3.2 Hybrid Cache Update

As our goal is to build a system that is robust for (nearly) arbitrary workloads, the limited forwarding rule update rate poses challenges for the caching mechanism. Since each cache addition or eviction requires a switch rule installation or removal, the rule update rate in switches directly limits the cache update rate, which affects how quickly SwitchKV can react to workload distribution changes. Though switches are continuously being optimized to speed up their rule update and some switches can now achieve 12K updates per second [33], they are still too slow to support traditional caching strategies that insert each recently-visited key to the cache.

To meet this new challenge, we designed new hybrid cache update algorithms and protocols to minimize unnecessary cache churn. The cache update mechanism consists of three components: 1) Backends *periodically* report recent hot keys to the cache nodes. 2) Backends *immediately* report keys that suddenly become very hot to the cache nodes. 3) Cache nodes add selected keys from reports and evict appropriate keys when necessary, and they instruct the network controller to make corresponding switch rule updates through REST APIs. Cache addition is prioritized over eviction in order to react quickly to sudden workload distribution changes at the cost of some additional buffer switch rule space. Fig. 4 shows our cache update mechanism at a high level.

3.2.1 Update with Periodic Hot Key Report

In most caching systems, a query for a key that is not in the cache would bring that key into cache and evict another key if the cache is full. However, many recently visited keys are not hot and will not be accessed again in the near future. This would result in unnecessary cache churn, which would harm the performance of SwitchKV because its cache update rate is limited.

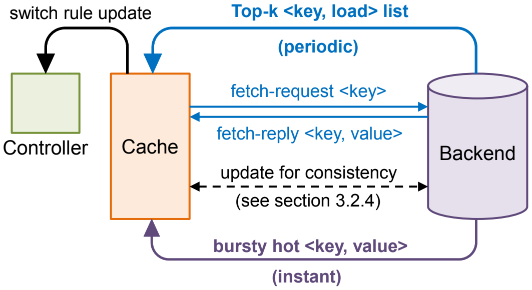


Figure 4: Cache update overview.

Instead, we use a different approach to add objects to the cache less aggressively. Each backend node maintains an efficient top-k load tracker to track recent popular keys. Backend nodes periodically (e.g., every second) report their recent hot keys and loads to the cache nodes. Each cache node maintains an in-memory data store and frequency counter for the cached items with the same load metric. The cache node keeps a load threshold based on the load statistics of cached keys. Upon receiving the reports, the cache node selects keys whose loads are above the threshold to add to the cache. It sends `fetch` requests for the selected keys to the corresponding backend nodes to get the values. It then updates the cache and instructs the network controller to update switch rules based on the received `fetch` responses.

Time-segmented Top-K Load Tracker. Each backend node maintains a key-load list with k entries to store its approximated hottest k keys and their loads. It also keeps a local frequency counter for the recently visited keys, so that it can know what are the most popular keys and how frequently they are queried. A backend node cannot afford to keep counters for all keys in memory. Instead, since only information about hot keys is needed, we can use memory-efficient top-k algorithms to find frequent items in the query stream [8].

To keep track of *recent* hot keys, we segment the query stream into separate intervals. At the end of each interval, the frequency counter extracts the top-k list of its current segment, then clears itself for the next segment. The key-load list is updated by the top-k list of the new segment using weighted average. Suppose the frequency of key x in the new segment is f_x, and the current load of x is L'_x, then the new load of x is

$$L_x = \alpha \cdot f_x + (1 - \alpha) \cdot L'_x, \tag{1}$$

where α represents the degree of weighting decrease. A higher α discounts previous load faster. Only keys in the new top-k list will be kept in the new updated key-load list. L'_x is zero for keys not in the previous key-load list.

Algorithm 1 Update Frequency Counter

```
function SeeQuery($x$)
    if $x$ is not tracked in the counter then
        if the counter is not full then
            create a bucket with $f = 1$ if not exists
            add $x$ to the first bucket;  return
        $y \leftarrow$ first key of first bucket, the least visited key
        replace $y$ with $x$ and keep the same frequency
    Update($x$)
function Update($x$)  // key $x$ is tracked in the counter
    $\langle b,\ f \rangle \leftarrow$ current $\langle$bucket, frequency$\rangle$ of $x$
    if next bucket of $b$ has frequency $f + 1$ then
        move $x$ to the next bucket
    else if $x$ is the only key of $b$ then
        increase frequency of $b$ to $f + 1$
    else move $x$ to a new bucket with frequency $f + 1$
    if $b$ is empty then delete $b$
```

The frequency counter uses a "space-saving algorithm" [30] to track the heavy hitters of the query stream in each time segment and approximate the frequencies of these keys. Fig. 5 shows the data structure of the frequency counter.

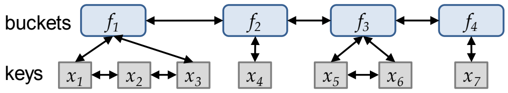


Figure 5: Structure of top-k frequency counter.

The counter consists of a linked list of buckets, each with a unique frequency number f. The buckets are sorted by their frequency in increasing order (e.g., $f_1 < f_2$). Each bucket has a linked list of keys that have been visited for the same number of times, f. Keys in the same bucket are sorted by their most recent visited time, with newest key at the tail of the list. With this structure, getting a list of top-k hot keys and their load is straightforward. For example, the top-5 list in Fig. 5 is $[\langle x_7, f_4\rangle\ \langle x_6, f_3\rangle\ \langle x_5, f_3\rangle\ \langle x_4, f_2\rangle\ \langle x_3, f_1\rangle]$.

The counter has a configurable size limit N, which is the maximum number of keys it can track. Algorithm 1 describes how to update the counter. When processing (e.g., create, delete, move) buckets and keys, the orders described above are always maintained. The counter requires $O(N)$ memory, and has $O(1)$ running time for each query. To reduce the computational overhead, we can randomly sample the query packets, and only update the counter for a small fraction of the queries. Sampling can provide a good approximation of the ranking of heavy hitters in highly skewed workloads.

Cache Adds Selected Keys from Reports. The cache also tracks the load for all cached keys. In order to be comparable with the load of reported keys, it must keep the same parameters (e.g., time segment interval, average weights, sampling rate) with the tracker in the backends. Cache nodes update a load threshold periodically based on the loads of cached keys, and send `fetch` queries for the reported keys with load higher than the threshold.

Too big of a threshold would prevent caching hot keys, while a too small of one would cause frequent unnecessary cache churn. To compute a proper load threshold in practice, the cache samples a certain number of key loads and uses the load at a certain rank (e.g., 10^{th} percentile from the lowest) as the threshold value. This process runs in the background periodically, so it does not introduce overhead to serving queries or updating cached data.

3.2.2 Update for Bursty Hot Keys

Periodic reports can update the cache effectively with low communication and memory overhead, but cannot react quickly when some keys suddenly become popular. In addition to periodic reports, the backends also send instant reports to the cache to report bursty queries, so that those queries can be offloaded to the cache immediately.

Each backend maintains a circular log to track the recently visited keys, and a hash table that keeps only entries for keys currently in the log and tracks the number of occurrence of these keys. As shown in Fig. 6, when a key is queried, it is inserted into the circular log, with the existing key at that position evicted. The hash table updates the count of the keys accordingly and adds or deletes related entries when necessary. If the count of a key exceeds a threshold and the node's overall load is also above a certain threshold, the key and its value are *immediately sent and added to the cache*. The size of the circular log and hash table could be small (e.g., a few hundreds of entries), which introduces little overhead to query processing.

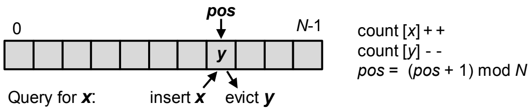


Figure 6: Circular log and counter.

3.2.3 Handle Burst Change with Rule Buffer

The distribution changes in real-world workloads are not constant. Sudden changes in the key popularity may lead a large number of cache updates in a short period of time. In traditional caching algorithms, a cache addition when the cache is full would also trigger a cache eviction, which in SwitchKV would mean that each addition involves two forwarding rule updates in the switch. As a result, the cache would only able to add keys at half of the switch update rate on average.

In order to react quickly to sudden workload changes, we prioritize cache addition over eviction. Cache evictions and switch rule deletion requests are queued and executed after cache additions and rule installations until a maximum delay is reached. In this way, we can

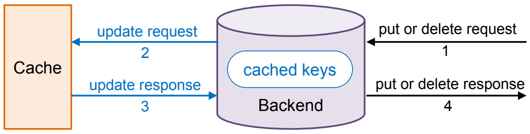


Figure 7: Updates to keep cache consistency.

reduce the required peak switch update rate for bursty cache updates to half, so that new hot keys can be added to cache more quickly. For example, if the switch update rate limit is 2000 rules per second, and the maximum delay for rule deletions is one second, then the cache can update at 1000 keys/second on average, and a maximum of 2000 keys/second for a short period (one second).

To allow delay in switch rule deletions, a rule buffer must be reserved in the L2 table. The size of this buffer is the maximum switch update rate times the duration of maximum delay. In the example above, the switch should reserve space for at least 2000 rules, which is small compared to the available L2 table size in switches.

Delaying rule deletion may result in stale forwarding rules in the L2 table. The stale rules will produce a temporary cache miss for some queries, as shown in the lower right block of Fig. 3. The miss overhead is small, however, because the evicted or deleted keys are (by definition) less likely to be frequently visited.

3.2.4 Cache Consistency

SwitchKV always guarantees consistent responses to clients. As a performance optimization, it allows temporary inconsistency between switch forwarding rules and cached keys, which (as described above) can introduce temporary overheads for a small number of queries, but never causes inconsistent data access.

In traditional cache systems such as Memcached [15], when a client sends a `put` or `delete` request, it will also send a request to the cache to either update or invalidate the item if it is in cache. The cache in SwitchKV is small and it is possible that most requests are for uncached keys, so forwarding each `put` or `delete` request to the cache introduces unnecessary overhead.

The backends avoid this overhead by tracking, in-memory, which keys in its local store are currently cached. The backend only updates the cache when it receives requests for one of these cached keys. Keys are added to the set whenever the backend receives a `fetch` request, or sends an instant hot object detected by the circular-log counter. When the cache evicts a key, or decides not to add the item from a `fetch` response or instant report, it sends a message to the backend so that the backend can remove this key from its cached key set.

We use standard leasing mechanisms to ensure consistency when there are cache or backend failures or network partitions [17]. Backends grant the cache a short-term lease on each cached key. The cache periodically renews its leases and only return a cached value while the lease is still valid. When a backend receives a `put` or `delete` request for a cached key, it will send an `update` request to the cache, as shown in Fig. 7, and will wait for the response or until the lease expires before it replies to the client. We choose to update the cached data rather than invalidate it for a `put` request to reduce the cache churn and rule update burden on switches.

3.3 Local Storage and Networking

Optimizing the single-node local performance of cache and backends is not our primary goal, and has been extensively researched [27, 28, 39]. Nevertheless, we made several design choices on local storage and networking to maximize the potential performance of each server, which we discuss here.

3.3.1 Parallel Data Access

Exploiting the parallelism of multi-core systems is critical for high performance. Many key-value systems use various concurrent data structures so that all cores can access the shared data in parallel [11, 14, 32]. However, they usually scale poorly with writes and can introduce significant overhead and complexity to our cache update algorithms that require query statistics tracking.

Instead, SwitchKV partitions the data in each cache and backend node based on key hash. Each core has *exclusive access* to its own partition, and runs its own load trackers. This greatly improves both the concurrency and simplicity of the local stores. Prior work [14, 29, 41] observed that partitioning may lower the performance when the load across partitions is imbalanced. In SwitchKV, however, backend nodes do not face high skew in key popularity. By exploiting CPU caches and packet burst I/O, a cache node that serves a small number of keys can handle different workload distributions [28].

3.3.2 Network Stack

SwitchKV uses Intel® DPDK [21] instead of standard socket I/O, which allows our user-level libraries to control NICs, modify packet headers, and transfer packet data with minimal overhead [28].

Since each core in the cache and backend nodes has exclusive access to its own partition of data, we can have the NIC deliver each query packet to the appropriate RX queue based on the key. SwitchKV can achieve this by using Receive Side Scaling (RSS) [12, 20] or Flow Director (FDir) [28, 36].[2] Both methods require information about the key in packet headers for the NIC to identify which RX queue should the packet be sent to. This requirement is automatic in SwitchKV where key hashes are already part of the packet header.

[2] Our prototype uses RSS. FDir enables more flexible control of the network stack, but it is not supported in the Mellanox NICs that we use.

3.4 Cluster Scaling

To scale system performance, the cluster will require multiple caches and OpenFlow switches. This section briefly sketches a design (not yet implemented) for a scale-out version of SwitchKV.

Multiple Caches. We can increase SwitchKV's total system throughput by deploying additional cache nodes. As each individual node can deliver high throughput because of its small dataset size (especially when keys fit within its L3 cache), we do not replicate keys across nodes and instead simply partition the cache across the set of participating nodes.[3] Each cache node is responsible for multiple backends, and each backend reports only to its dedicated cache node. As such, we do not require any cache coherency protocols between the cache nodes.

If the mapping between backends and cache nodes changes, the relevant backends will delete their cached items from their old cache nodes, and then report to the new ones. If the change is due to a cache crash, the network controller will detect the failed node and delete all forwarding rules to it.

Network Scaling. To scale network throughput, we can use the well-studied multi-rooted fat-tree [1, 19]. Such an architecture may require exact match rules for cached keys to be replicated at multiple switches. This approach may sacrifice performance until the rule updates complete, but does not compromise correctness (the backends may need to serve the keys temporarily).

On the other hand, if the switching bottleneck is in terms of rule space (as opposed to bandwidth), then each switch must be configured to store only rules for a subset of the backend nodes, i.e., we partition the backends, and thus the rule space, across our switches. In this case, queries for keys in a backend node must be sent through a switch associated with that key's backend (i.e., that has the appropriate rules); that switch can be identified easily by the query packets' destination IP addresses.

4 Evaluation

In this section, we demonstrate how our new architecture and algorithms significantly improve the overall performance of a key-value storage cluster under various workloads. Our experiments answer three questions:

- How well does a fast small cache improve the cluster load balance and overall throughput? (§4.2)
- Does SwitchKV improve system throughput and latency compared to traditional architectures? (§4.3)
- Can SwitchKV's new cache update mechanism react quickly to workload changes? (§4.4)

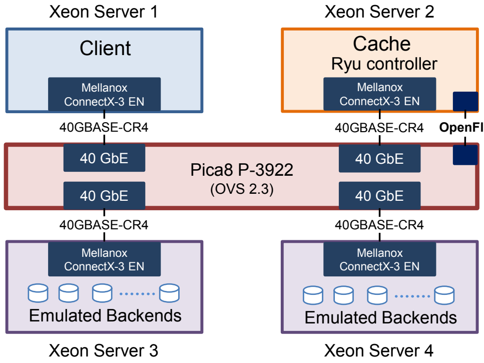


Figure 8: Evaluation platform.

Our SwitchKV prototype is written in C/C++ and runs on x86-64 Linux. Packet I/O uses DPDK 2.0 [21]. In order to minimize the effects of implementation (rather than architectural) differences, we implemented the look-aside and look-through caches used in our evaluation simply by changing the query data path in SwitchKV.

4.1 Evaluation Setup

Platform. Our testbed consists of four server machines and one OpenFlow switch. Each machine is equipped with dual 8-core CPUs (Intel® Xeon® E5-2660 processors @ 2.20 GHz), 32 GB of total system memory, and one 40Gb Ethernet port (Mellanox ConnectX-3 EN) that is connected to one of the four 40GbE ports on a Pica8 P-3922 switch. Fig. 8 diagrams our evaluation platform. One machine serves as the client, one machine as the cache, and two machines emulate many backends nodes.

We derived our emulated performance from experimental measurements on a backend node that fits our target configuration: an Intel® Atom™ C2750 processor paired with an Intel® DC P3600 PCIe-based SSD. On this SSD-based target backend, we ran RocksDB [39] with 120 million 1KB key-value pairs, and measured its performance against a client over a 1Gb link. The backend could serve 99.4K queries per second on average.

Each emulated backend node in our experiments runs its own isolated in-memory data structures to serve queries, track workloads, and update the cache. It has a configurable maximum throughput enforced by a fine-grained rate limiter.[4] The emulated backends do not store the actual key-value pairs due to limited memory space. Instead, they reply to the client or update the cache with a fake random value for each key. In most experiments (except Fig. 14), we emulate a total of 128 backend nodes in the two server machines, and limit each node to serve at most 100K queries per second. Table 2 summarize the default experiment settings unless otherwise specified.

[3] Note that while we *are* very concerned about load amongst our backend nodes, our cache nodes have orders-of-magnitude higher performance, and thus the same load-balancing concerns do not arise.

[4] Since it is hard to predict the performance bottleneck at a backend node if its load is skewed, we assume backends have a fixed throughput limit as measured under uniform workloads.

Number of backend nodes	128
Max throughput of each backend	100 KQPS
Workload distribution	Zipf (0.99)
Number of items in cache	10000

Table 2: Default experiment settings unless otherwise specified

Workloads and Method. We evaluate both skewed and uniform workloads in our experiments, and focus mainly on skewed workloads. Most skewed workloads use a non-uniform key popularity that follows a Zipf distribution of skewness 0.99, which is the same that used by YCSB [7]. The request generator uses approximation techniques to quickly generate workloads with a Zipf distribution [18, 28]. The keyspace size is 10 billion, so each of the 128 backend nodes is responsible for serving approximately 78 million unique keys. The mapping of a given key to a backend is decided by the key hash. We use fixed 16-byte keys and 128-byte values.

Most experiments (except Fig. 12) use read-only workloads, since SwitchKV aims to load balance read requests. All write requests have to be processed by the backends, so they cannot be load balanced by the cache.

To find the maximum effective system throughput, the client tracks the packet loss rate, and adjusts its sending rate every 10 milliseconds to keep the loss rate between 0.5% to 1%. This self-adjusted rate control enables us to evaluate the real-time system performance.

Our server machines can send packets at 28 Mpps, but receive at only 15 Mpps. To avoid the system being bottlenecked by the client's receiving rate, the backends and cache node fully process all incoming queries, but send only half of the responses back to the client. The client doubles its receiving rate before computing the loss rate.

4.2 Load Balancing with a Small Cache

We first evaluate the effectiveness of introducing a small cache for reducing load imbalances.

Fig. 9 shows a snapshot of the individual backend node throughput with caching disabled under workloads of varying skewness. We observe that the load across the backend nodes is highly imbalanced.

Fig. 10 shows how caching affects the system throughput. Under uniform random workload, the backends total throughput can reach near the maximum capacity (128 backends × 100 KQPS). However, when the workload is skewed, the system throughput without the cache is bottlenecked by the overloaded node and significantly reduced. Adding a small cache can help the system achieve good load balance across all of nodes: A cache with only 10,000 items can improve the system's overall throughput by 7× for workloads with Zipf skewness of 0.99.

Fig. 11 investigates how different numbers of cached items affect the system throughput. The backends' load quickly becomes well balanced as the number of cached items grows to 1000. Then, the system throughput continues to grow as more items are cached, but the benefits from increased cache size diminish (as one expects given a Zipf workload). The system would require significantly more memory at the cache node or many more cache nodes to further increase the hit ratio. We choose to cache 10,000 items for the rest of the experiments.

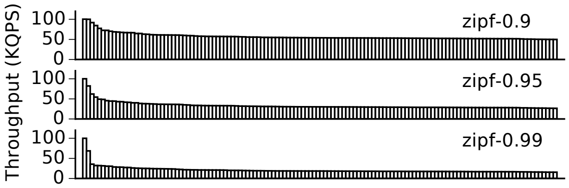


Figure 9: Throughput of each backend node without cache under workloads with different Zipf skewness. Node IDs (x-axis) are sorted according to their throughput.

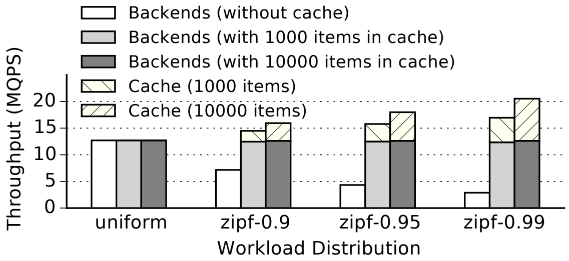


Figure 10: System throughput with and without the use of a cache. Figure illustrates the portion of total throughput handled by the cache and that by backend nodes.

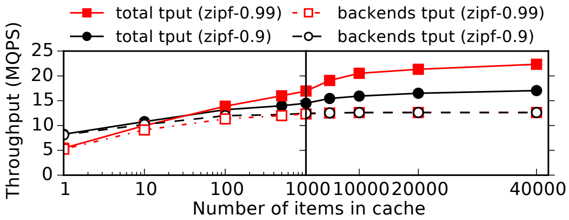


Figure 11: System throughput as cache size increases. Even a modest-sized cache of 10,000 items achieves significant gains.

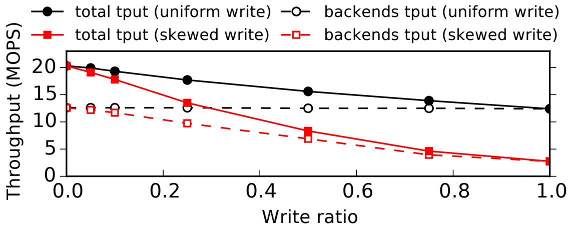
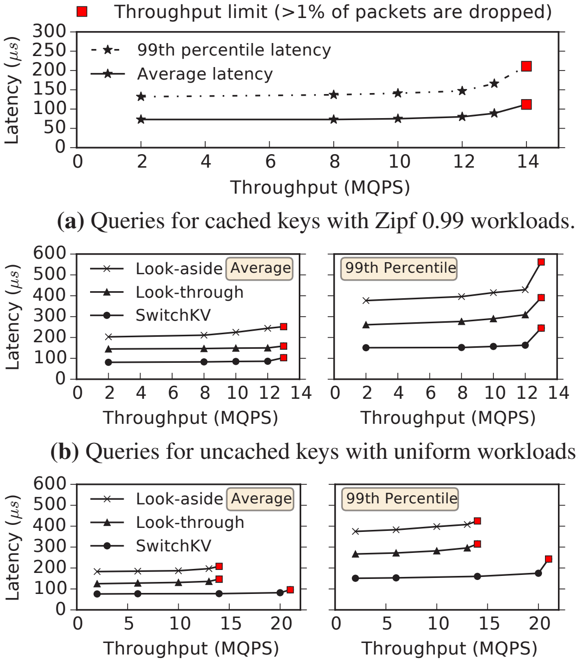


Figure 12: System throughput with different write ratio.

Fig. 12 plots the systems throughput with different write ratios and write workloads. We assume the backend nodes have the same performance for read and write operations, and use two types of write workload: write queries uniformly distributed across all keys and write queries according to the same Zipf 0.99 distribution as read queries. Write workloads cannot be balanced by



(a) Queries for cached keys with Zipf 0.99 workloads.

(b) Queries for uncached keys with uniform workloads.

(c) Zipf 0.99 workloads with 10000 items in cache.

Figure 13: End-to-end latency as a function of throughput.

the cache, so the system throughput with skewed write workload quickly decreases as the write ratio increases. With the uniform write workload, load across the backends is always uniform, so increasing the write ratio only decreases the effective throughput of the cache.

4.3 Benefits of the New Architecture

This section compares the system performance between SwitchKV and traditional look-aside and on-path look-through architectures. As summarized in Table 1, compared to traditional architectures in which the cache handles all queries first, the cache in SwitchKV is only involved when the requested key is already cached (with high likelihood), and thus uncached items are served with only a single machine transit. As a result, we expect SwitchKV to have both lower latency and higher throughput than traditional architectures, which is strongly supported by our experimental results.

Latency. We first compare the average and 99^{th} percentile latency of different architectures, as shown in Fig. 13. To measure the end-to-end latency, the client tags each query packet with the current timestamp. When receiving responses, the client compares the current timestamp and the previous timestamp echoed back in the responses. To measure latency under different throughputs, we disable the client's self rate adjustment, and manually set different send rates.

Fig. 13a shows the latency when the client only sends queries for keys in the cache. In all three architectures,

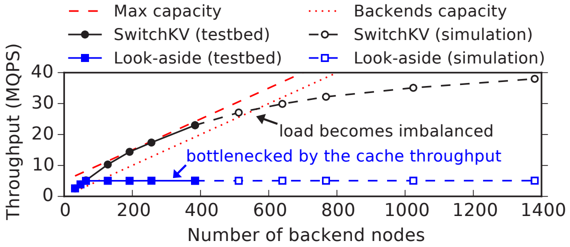


Figure 14: System throughput scalability as the number of backend nodes increases, for SwitchKV and look-aside architecture with Zipf 0.99 workload and at most 10000 items in cache. On-path look-through has the same throughput as look-aside. Each backend node is rate limited at 50K queries per second, cache is rate limited at 5 million queries per second. Look-through has similar performance to look-aside.

the queries will be forwarded to the cache by the switch and the cache reply directly to the client. Accordingly, they have the same latency for cache hits.

Fig. 13b shows the latency when the client generates uniform workloads and the cache is empty, which results in all queries missing the cache. Look-aside has the highest latency because it takes three machine transits (cache→client→backend) to handle a cache miss. Look-through also has high latency because it takes two machine transits (cache→backend) to handle a cache miss. In comparison, queries for uncached keys in SwitchKV cache will directly go to the backend nodes.

Fig. 13c shows the overall latency for a Zipf 0.99 workload and 10000-item cache. As shown in Fig. 10, about 38% of queries will hit the cache under these settings. The average latency is within the range of cache hits and cache misses. The 99^{th} percentile latency is about the same as cache miss latency. As all queries must go through the cache in look-aside and look-through architectures, we cannot collect latency measurements beyond the 14 million QPS mark for them, as the cache is unable to handle more traffic. This result illustrates one of the major benefits of the SwitchKV design: requests for uncached keys are simply not sent to the cache, allowing a single cache node to support more backends (higher aggregate system throughput).

Throughput. We then compare the full system throughput under a Zipf 0.99 workload as the number of backend nodes increase, for different architectures. For each architecture, the cache node stores at most 10000 items.

In order to emulate more backend nodes in this experiment, we scale down the rate capacity of each backend node to at most 50K queries per second, and limit the cache to serve at most 5 million queries per second. The performance improvement ratio of SwitchKV to other architectures will be the same as long as the performance ratio of the cache to a backend node is 100:1. To achieve the maximum system throughput, the cache may store fewer items when it becomes the performance bottleneck as the backend cluster size increases.

Fig. 14 shows the experiment results. The throughput of the look-aside architecture is bottlenecked quickly by the cache capacity when the number of backend nodes increases to 64, while the throughput of SwitchKV can scale out to much larger cluster sizes. When the number of backend nodes goes beyond 400, the throughput begins to drop below the maximum system capacity, because the cache is insufficient for providing good load balance for such a cluster. To retain linear scalability as the cluster grows, we would need to have a more powerful cache node or increase the number of cache nodes.

Less skewed workloads will yield better scalability for SwitchKV, but will hit the same performance bottleneck for both look-aside and look-through architectures. Due to space constraints, we omit these results.

4.4 Cache Updates

This section evaluates the effectiveness of SwitchKV's hybrid cache-update mechanisms. In these experiments, we keep the workload distribution (Zipf 0.99) the same, and change only the popularity of each key. The workload generator in the client actually generates key indices with fixed popularity ranks. We change the query workloads by changing the mapping between indices and key strings. We use three different workload change patterns:

1. ***Hot-in***: Move N cold keys to the top of the popularity ranks, and decrease the ranks of other keys accordingly. This change is radical, as cold keys suddenly become the hottest ones in the cluster.
2. ***Hot-out***: Move N hottest keys to the bottom of the popularity ranks, and increase the ranks of other keys accordingly. This change is more moderate, since the new hottest keys are most likely already in the cache if N is smaller than the cache size.
3. ***Random***: Replace N random keys in the top K hottest keys with cold keys. We typically set K to the cache size. This change is typically moderate when N is not large, since the probability that most of the hottest keys are changed at once is low.

A note about our experimental infrastructure, which affects SwitchKV's performance under rapid workload changes: The Pica8 P-3922 switch's L2 rule update is poorly implemented. The switch performs an unnecessary linear scan of all existing rules before each rule installation, which makes the updates very slow as the L2 table grows. We benchmark the switch and find it can only update about 400 rules/second when the there are about 10K existing rules, which means the cache can only update 200 items/second on average. Some other switches can update their rules much faster (e.g., 12K updates/second [33]). Though still too slow to support the update rate needed by traditional caching algorithms, these switches would provide much higher performance with SwitchKV under rapidly changing workloads.

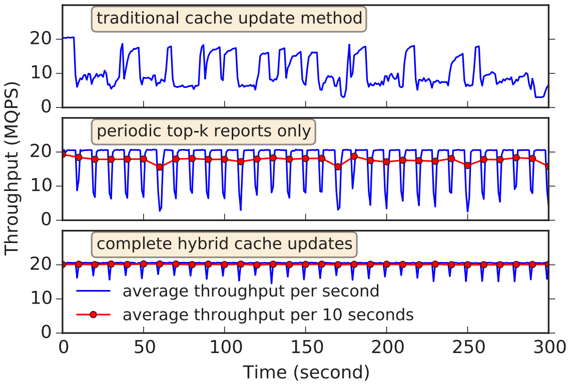


Figure 15: Throughput with *hot-in* workload changes, i.e., change 200 cold keys into the hottest keys every 10 seconds.

All experiments use Zipf 0.99 workloads and a 10000-item-sized cache. Each experiment begins with a prepopulated cache containing the top 10,000 hot items. Each backend node sends reports to the cache as follows: its top five hot keys every second, and keys that were visited more than eight times within the last two hundred queries instantly. The choice of parameters for periodic and instant updates is flexible, determined by the performance goals, cache size, and update rate limit. For example, the size and threshold of the ring counter for instant reports determines when a key is hot enough to be immediately added to the cache. A threshold that is too low may cause unnecessary cache churn, while a threshold that is too high may make the cache slow to respond to bursty workload changes. We omit a sensitivity analysis of these parameters due to space limits. We also compare SwitchKV with a traditional update method, in which backends try to add every queried key to the cache.

We first evaluate system throughput under the *hot-in* change pattern. Since this is a radical change, we do not expect it to happen frequently. Thus, we move 200 cold keys to the top of the popularity ranks every ten seconds. Fig. 15 shows the system throughput over time. A traditional cache update method has very poor performance, as it performs many cache updates for recently-visited yet non-hot keys. With periodic top-k reports alone, a backend's hot keys are not added to the cache until its next report (once per second). The throughput is reduced to less than half after the workload changes, and recovers in 1-2 seconds. The bottom subfigure shows SwitchKV's throughput using its complete cache update mechanism, which includes the instant hot key reports. The new hot keys are immediately added to the cache, resulting in a lower performance drop and a much faster recovery after a sudden workload change. This demonstrates that SwitchKV is robust enough to meet the SLOs even with certain adversarial changes in key popularity.

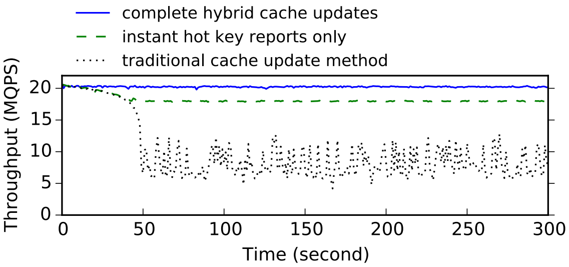


Figure 16: Throughput with *hot-out* workload changes, i.e., move out 200 hottest keys every second.

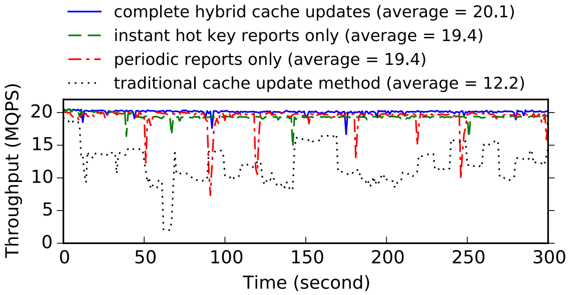


Figure 17: Throughput with *random* workload changes, i.e., replace 200 out of the top 10000 keys every second.

Our next experiment evaluates SwitchKV's throughput under a *hot-out* change pattern. Every second, the 200 hottest keys suddenly go cold, and we thus increase the popularity ranks of all other keys accordingly. As shown in Fig. 16, the complete update mechanism can handle this change well. With instant reports only and no periodic reports, the system cannot achieve its maximum throughput: the circular log counter can detect only very hot keys, not the keys just entering the bottom of the top-10000 hot-key list. These keys are only added to cache as they further increase in their popularity when more of the hottest keys move out. Note that this gap becomes particularly apparent as the system reaches its steady state 50 seconds into the experiment; at this point, none of the pre-populated cached keys remain in the cache.

Fig. 17 shows the throughput with a *random* change pattern, in which we randomly replace 200 keys in the top 10000 popular keys every second. The complete update mechanism is able to handle the workload changes. There are occasionally short-term small performance drops, which occur when the hottest keys are replaced. The throughput would be lower, however, if SwitchKV were to omit either its instant or periodic reports.

Fig 18 shows the effectiveness of SwitchKV's rule buffer in handling bursty workload changes (see §3.2.3). The maximum delay for cache eviction and rule deletion is set to 2 seconds. With a switch rule buffer and prioritizing rule installation, the 600 new keys can be added to the cache within 1.5 seconds. Without the rule buffer, this installation time would double. The rule buffer thus reduces any throughput impact and allows faster recovery during bursty workload changes.

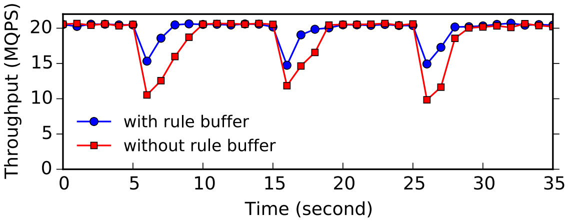


Figure 18: Throughput with *hot-in* workload changes with 600 new hottest keys every time, which requires 1200 rule updates and will take the switch at least three seconds to finish them.

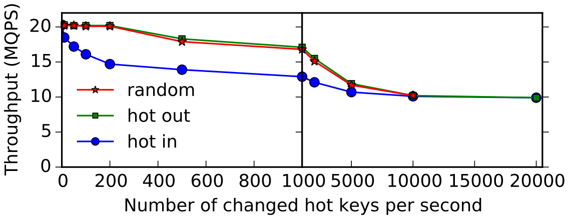


Figure 19: Throughput with different workload change patterns as a function of change rate.

Fig. 19 shows the average throughput with different change patterns and rates. The switch can update 400 rules per second, which can support 200 cache updates per second. The system throughput is near maximum for random and hot-out change patterns when the change rate is within 200 keys per second, and then goes down as the change rate increases. Throughput drops quickly under increasing hot-in changes, as the cache is less effective when more of the hottest keys change every second. Once all patterns change more than 10000 of the hottest keys per second, all three patterns yield similar throughput, as all patterns replace the entire cache every second. Still, even at this point the cache can still keep up to 200 of current hot keys, and most of the hottest keys are likely to added to the cache from the instant reports, so throughput is still much higher (by 3×) than that of the system lacking a cache. The performance under fast changing workloads would be higher with switches that can update their rules faster.

5 Conclusion

SwitchKV is a scalable key-value storage system that can maintain efficient load balancing under widely varying and rapidly changing real-world workloads. It achieves high performance in a cost effective manner, both by combining fast small caches with new algorithm design, and by exploiting SDN techniques and switch hardware. We demonstrate SwitchKV can meet the service-level objectives for throughput and latency more efficiently than traditional systems.

Acknowledgments

The authors are grateful to Hyeontaek Lim, Anuj Kalia, Sol Boucher, Conglong Li, Nanxi Kang, Xin Jin, Linpeng Tang, Aaron Blankstein, Haoyu Zhang, our shepherd Jinyang Li and the anonymous NSDI reviewers for their constructive feedback. This work was supported by National Science Foundation Awards CSR-0953197 (CAREER) and CCF-0964474, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2008.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012.

[4] A. Bestavros. Www traffic reduction and load balancing through server-based caching. *IEEE Parallel Distrib. Technol.*, 5(1), Jan. 1997.

[5] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*, 2010.

[6] Y. Cheng, A. Gupta, and A. R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010.

[8] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 1(2), Aug. 2008.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[10] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2), Feb. 2013.

[11] B. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2), Sept. 2010.

[12] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[13] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, 2011.

[14] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[15] B. Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004(124):5–, Aug. 2004. ISSN 1075-3583.

[16] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2011.

[17] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP)*, 1989.

[18] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 1994.

[19] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2009.

[20] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM Conference*, 2010.

[21] Intel Data Plane Development Kit (DPDK). `http://dpdk.org/`.

[22] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In *Proceedings of the 11th International Conference on World Wide Web (WWW)*, 2002.

[23] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC)*, 1997.

[24] M. Klems, A. Silberstein, J. Chen, M. Mortazavi, S. A. Albert, P. Narayan, A. Tumbde, and B. Cooper. The yahoo!: Cloud datastore load balancer. In *Proceedings of the Fourth International Workshop on Cloud Data Management (CloudDB)*, 2012.

[25] D. Kreutz, F. Ramos, P. Esteves Verissimo,

C. Ẽsteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1), Jan. 2015.

[26] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015.

[27] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[28] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014.

[29] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[30] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory (ICDT)*, 2005.

[31] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10), Oct. 2001.

[32] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[33] NoviSwitch. `http://noviflow.com/products/noviswitch/`.

[34] NVM Express. `http://www.nvmexpress.org/`.

[35] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, Jan. 2010.

[36] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[37] Pica8. `http://www.pica8.com/`.

[38] Redis. `http://redis.io/`.

[39] RocksDB. `http://rocksdb.org/`.

[40] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3), Nov. 2014.

[41] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

Bitcoin-NG: A Scalable Blockchain Protocol[*]

Ittay Eyal Adem Efe Gencer Emin Gün Sirer Robbert van Renesse
Cornell University

Abstract

Cryptocurrencies, based on and led by Bitcoin, have shown promise as infrastructure for pseudonymous online payments, cheap remittance, trustless digital asset exchange, and smart contracts. However, Bitcoin-derived blockchain protocols have inherent scalability limits that trade off between throughput and latency, which withhold the realization of this potential.

This paper presents Bitcoin-NG (Next Generation), a new blockchain protocol designed to scale. Bitcoin-NG is a Byzantine fault tolerant blockchain protocol that is robust to extreme churn and shares the same trust model as Bitcoin.

In addition to Bitcoin-NG, we introduce several novel metrics of interest in quantifying the security and efficiency of Bitcoin-like blockchain protocols. We implement Bitcoin-NG and perform large-scale experiments at 15% the size of the operational Bitcoin system, using unchanged clients of both protocols. These experiments demonstrate that Bitcoin-NG scales optimally, with bandwidth limited only by the capacity of the individual nodes and latency limited only by the propagation time of the network.

1 Introduction

Bitcoin has emerged as the first widely-deployed, decentralized global currency, and sparked hundreds of copycat currencies. Overall, cryptocurrencies have garnered much attention from the financial and tech sectors, as well as academics; achieved wide market penetration in underground economies [38]; reached a $12B market cap; and attracted close to $1B in venture capital [15]. The core technological innovation powering these systems is the *Nakamoto consensus* protocol for maintaining a distributed ledger known as the blockchain. The blockchain technology provides a decentralized, open, Byzantine fault-tolerant transaction mechanism, and promises to become the infrastructure for a new generation of Internet interaction, including anonymous online payments [14], remittance, and transaction of digital assets [16]. Ongoing work explores smart digital contracts, enabling anonymous parties to programmatically enforce complex agreements [31, 56].

Despite its potential, blockchain protocols face a significant scalability barrier [51, 36, 19, 5]. The maximum rate at which these systems can process transactions is capped by the choice of two parameters: block size and block interval. Increasing block size improves throughput, but the resulting bigger blocks take longer to propagate in the network. Reducing the block interval reduces latency, but leads to instability where the system is in disagreement and the blockchain is subject to reorganization. Bitcoin currently targets a conservative 10 minutes between blocks, yielding 10-minute expected latencies for transactions to be encoded in the blockchain. The block size is currently set at 1MB, yielding only 1 to 3.5 transactions per second for Bitcoin for typical transaction sizes. Proposals for increasing the block size are the topic of heated debate within the Bitcoin community [47].

In this paper, we present Bitcoin-NG, a scalable blockchain protocol, based on the same trust model as Bitcoin. Bitcoin-NG's latency is limited only by the propagation delay of the network, and its bandwidth is limited only by the processing capacity of the individual nodes. Bitcoin-NG achieves this performance improvement by decoupling Bitcoin's blockchain operation into two planes: *leader election* and *transaction serialization*. It divides time into epochs, where each epoch has a single leader. As in Bitcoin, leader election is performed

[*] The authors are supported in part by AFOSR grants FA2386-12-1-3008, F9550-06-0019, by the AFOSR MURI Science of Cyber Security: Modeling, Composition, and Measurement as AFOSR grant FA9550-11-1-0137, by NSF grants CNS-1601879, 0430161, 0964409, 1040689, 1047540, 1518779, 1561209, and CCF-0424422 (TRUST), by ONR grants N00014-01-1-0968 and N00014-09-1-0652, by DARPA grants FA8750-10-2-0238 and FA8750-11-2-0256, by MDCN/iAd grant 54083, and by grants from Microsoft Corporation, Facebook Inc., and Amazon.com.

randomly and infrequently. Once a leader is chosen, it is entitled to serialize transactions unilaterally until a new leader is chosen, marking the end of the former's epoch.

While this approach is a significant departure from Bitcoin's operation, Bitcoin-NG maintains Bitcoin's security properties. Implicitly, leader election is already taking place in Bitcoin. But in Bitcoin, the leader is in charge of serializing history, making the entire duration of time between leader elections a long system freeze. In contrast, leader election in Bitcoin-NG is forward-looking, and ensures that the system is able to continually process transactions.

Evaluating the performance and functionality of new consensus protocols is a challenging task. To help perform this quantitatively and provide a foundation for the comparison of alternative consensus protocols, we introduce several metrics to evaluate implementations of Nakamoto consensus. These metrics capture performance metrics such as protocol goodput and latency, as well as various aspects of its security, including its ability to maintain consensus and resist centralization.

We evaluate the performance of Bitcoin-NG on a large emulation testbed consisting of 1000 nodes, amounting to over 15% of the current operational Bitcoin network [41]. This testbed enables us to run unchanged clients, using realistic Internet latencies. We compare Bitcoin-NG with the original Bitcoin client, and demonstrate the critical trade-offs inherent in the original Bitcoin protocol. Controlling for network bandwidth, reducing Bitcoin's latency by decreasing the block interval and improving its throughput by increasing the block size both yield adverse effects. In particular, fairness suffers, giving large miners an advantage over small miners. This anomaly leads to centralization, where the mining power tends to be concentrated under a single controller, breaking the basic premise of the decentralized cryptocurrency vision. Additionally, mining power is lost, making the system more vulnerable to attacks. In contrast, Bitcoin-NG improves latency and throughput to the maximum allowed by network conditions and node processing limits, while avoiding the fairness and mining power utilization problems.

In summary, this paper makes three contributions. First, it outlines the Bitcoin-NG scalable blockchain protocol, which achieves significantly higher throughput and lower latency than Bitcoin while maintaining the Bitcoin trust assumptions. Second, it introduces quantitative metrics for evaluating Nakamoto consensus protocols. These metrics are designed to ground the ongoing discussion over parameter selection in Bitcoin-derived currency. Finally, it quantifies, through large-scale experiments, Bitcoin-NG's robustness and scalability.

2 Model and Goal

The system is comprised of a set of nodes $\mathscr{N}$ connected by a reliable peer-to-peer network. Each node can poll a random oracle [6] as a random bit source. Nodes can generate key-pairs, but there is no trusted public key infrastructure.

The system employs a cryptopuzzle system, defined by a cryptographic hash function H. The solution to a puzzle defined by the string y is a string x such that $H(y|x)$ — the hash of the concatenation of the two — is smaller than some target. Each node i has a limited amount of compute power, called *mining power*, measured by the number of potential puzzle solutions it can try per second. A solution to a puzzle constitutes a *proof of work*, as it statistically indicates the amount of work a node had to perform in order to find it.

At any time t, a subset of nodes $B(t) \subset \mathscr{N}$ are Byzantine and behave arbitrarily, controlled by a single adversary. The other nodes are *honest* — they abide by the protocol. The mining power of each node i is $m(i)$. The mining power of the Byzantine nodes is less than 1/4 of the total compute power at any given time:

$$\forall t: \sum_{b \in B(t)} m(b) < \frac{1}{4} \sum_{n \in \mathscr{N}} m(n)$$

because proof-of-work blockchains, Bitcoin-NG included, are vulnerable to selfish mining by attackers larger than $1/4$ of the network [25].

Nakamoto Consensus

The nodes are to implement a replicated state machine (RSM) [33, 50]. Properties of the system can be compared to those of classical consensus [46]:

Termination There exists a time difference function $\Delta(\cdot)$ such that, given a time t and a value $0 < \varepsilon < 1$, the probability is smaller than ε that at times $t', t'' > t + \Delta(\varepsilon)$ a node returns two different states for the machine at time t.

Agreement There exists a time difference function $\Delta(\cdot)$ such that, given a $0 < \varepsilon < 1$, the probability that at time t two nodes return different states for $t - \Delta(\varepsilon)$ is smaller than ε.

Validity If the fraction of mining power of Byzantine nodes is bounded by f, i.e., $\forall t : \frac{\sum_{b \in B(t)} m(b)}{\sum_{n \in \mathscr{N}} m(n)} < f$, then the average fraction of state machine transitions that are not inputs of honest nodes is smaller than f.

3 Bitcoin and its Blockchain Protocol

Bitcoin is a distributed, decentralized crypto-currency [7, 8, 9, 43], which implicitly defined and implemented

Nakamoto consensus. Bitcoin uses the blockchain protocol to serialize transactions of the Bitcoin currency among its users. The replicated state machine maintains the balances of the different users, and its transitions are transactions that move funds among them. This state machine is managed by the system nodes, called miners.

Each user commands *addresses*, and sends Bitcoins by forming a transaction from her address to another's address and sending it to the nodes. More explicitly, a transaction is from the output of a previous transaction to a specific address. An output is *spent* if it is the input of another transaction. A client owns x Bitcoins at time t if the aggregate of unspent outputs to its address is x. Transactions are protected with cryptographic techniques that ensure only the rightful owner of a Bitcoin address can transfer funds from it. Miners accept transactions only if their sources have not been spent, thereby preventing users from double-spending their funds. The miners commit the transactions into a global append-only log called the *blockchain*.

The blockchain records transactions in units of blocks. Each block includes a unique ID, and the ID of the preceding block. The first block, dubbed *the genesis block*, is defined as part of the protocol. A valid block contains (1) a solution to a cryptopuzzle involving the hash of the previous block, (2) the hash (specifically, the Merkle root) of the transactions in the current block, which have to be valid, and (3) a special transaction, called the *coinbase*, crediting the miner with the reward for solving the cryptopuzzle. This process is called Bitcoin *mining*, and, by slight abuse of terminology, we refer to the creation of blocks as *block mining*. The specific cryptopuzzle is a double-hash of the block header whose result has to be smaller than a set value. The *problem difficulty*, set by this value, is dynamically adjusted such that blocks are generated at an average rate of one every ten minutes.

Mining When a miner creates a block, she is compensated for her efforts with Bitcoins. This compensation includes a per-transaction fee paid by the users whose transactions are included, as well as an amount of new Bitcoins that did not exist before.

Forks Any miner may add a valid block to the chain by simply publishing it over an overlay network to all other miners. If multiple miners create blocks with the same preceding block, the chain is *forked* into *branches*, forming a tree. Other miners may subsequently add new valid blocks to any of these branches. When a miner tries to add a new block after an existing block, we say it *mines on* the existing block. If this block is a leaf of a branch, we say he mines on the branch.

To resolve forks, the protocol prescribes on which chain the miners should mine. The criterion is that the winning chain is the *heaviest one*, that is, the one that required (in expectancy) the most mining power to generate. All miners add blocks to the heaviest chain of which they know, with random tie-breaking. We note that choosing a longest branch at random is suggested by Eyal and Sirer [25]. The operational client currently chooses the first branch it has heard of, making it more vulnerable in the general case. The heaviest chain a node knows is the serialization of RSM inputs it knows, and hence describes the RSM's state. The formation of forks is undesirable, as they indicate that there is no globally-agreed RSM state.

Branches and blocks outside the main chain are called pruned (and not orphans, as is common in informal discussions, since they have a parent in the block tree). Transactions in pruned blocks are ignored. They can be placed in the main chain at any later time, unless a contradicting transaction (that spends the same outputs) was placed there in the meantime.

Block dissemination over the Bitcoin overlay network takes seconds, whereas the average mining interval is ten minutes. Therefore, accidental bifurcation occurs on average about once every 60 blocks [18].

We are now ready to describe Bitcoin-NG.

4 Bitcoin-NG

Bitcoin-NG is a blockchain protocol that serializes transactions, much like Bitcoin, but allows for better latency and bandwidth without sacrificing other properties.

The protocol divides time into epochs. In each epoch, a single leader is in charge of serializing state machine transitions. To facilitate state propagation, leaders generate blocks. The protocol introduces two types of blocks: *key blocks* for leader election and *microblocks* that contain the ledger entries. Each block has a header that contains, among other fields, the unique reference of its predecessor; namely, a cryptographic hash of the predecessor header.

We detail the operation of the protocol in this section and explain its incentive system in Section 5.

4.1 Key Blocks and Leader Election

Key blocks are used to choose a leader. Like a Bitcoin block, a key block contains the reference to the previous block (either a key block or a microblock, usually the latter), the current Unix time, a coinbase transaction to pay out the reward, a target value, and a nonce field containing arbitrary bits. As in Bitcoin, for a key block to be valid, the cryptographic hash of its header must be smaller than the target value. Unlike Bitcoin, a key block contains a public key that will be used in subsequent microblocks.

As in Bitcoin, for a miner to generate a key block, it must iterate through nonce values until the crypto-puzzle condition is met. Consequently, the interval between

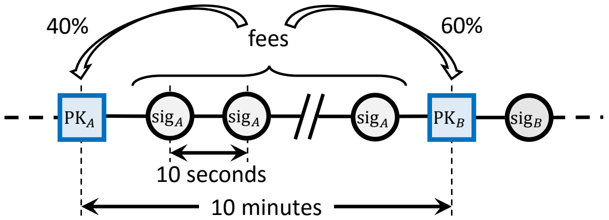


Figure 1: Structure of the Bitcoin-NG chain. Microblocks (circles) are signed with the private key matching the public key in the last key block (squares). Fee is distributed 40% to the leader and 60% to the next one.

consecutive key blocks is exponentially distributed. To maintain a set average rate, the difficulty is adjusted by deterministically changing the target value based on the Unix time in the key block headers.

In case of a fork, just as in Bitcoin, the nodes pick the branch with the most work, aggregated over all key blocks, with random tie breaking.

4.2 Microblocks

Once a node generates a key block it becomes the leader. As a leader, the node is allowed to generate microblocks at a set rate smaller than a predefined maximum. The maximum rate is deterministic, and can be much higher than the average interval between key blocks. The size of microblocks is bounded by a predefined maximum. Specifically, if the timestamp of a microblock is in the future, or if its difference with its predecessor's timestamp is smaller than the minimum, then the microblock is invalid. This bound prohibits a leader (malicious, greedy, or broken) from swamping the system with microblocks.

A microblock contains ledger entries and a header. The header contains the reference to the previous block, the current Unix time, a cryptographic hash of its ledger entries, and a cryptographic signature of the header. The signature uses the private key that matches the public key in the latest key block in the chain. For a microblock to be valid, all its entries must be valid according to the specification of the state machine, and the signature has to be valid. Figure 1 illustrates the structure.

Note that microblocks do not affect the weight of the chain, as they do not contain proof of work. This is critical for keeping the incentives aligned, as explained in Section 5.

4.3 Confirmation Time

When a miner generates a key block, he may not have heard of all microblocks generated by the previous leader. If microblock generation is frequent, this can be the common case on leader switching. The result is a short microblock fork, as illustrated in Figure 2. Such a fork is observed by any node that receives the to-be-pruned microblock (blocks A_3 and A_4 in the figure) before the new key block (block B in the figure). It is resolved once the key block propagates to that node. Therefore, a user that sees a microblock should wait for the propagation time of the network before considering it in the chain, to make sure it is not pruned by a new key block.

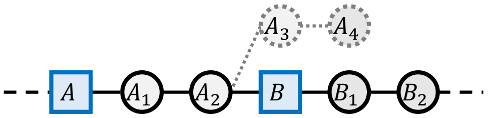


Figure 2: When microblocks are frequent, short forks occur on almost every leader switch.

4.4 Remuneration

To motivate mining, a leader is compensated for her efforts by the protocol. Remuneration is comprised of two parts. First, each key block entitles its generator a set amount. Second, each ledger entry carries a fee. This fee is split by the leader that places this entry in a microblock, and the subsequent leader that generates the next key block. Specifically, the current leader earns 40% of the fee, and the subsequent leader earns 60% of the fee, as illustrated in Figure 1. The choice of this distribution is explained in Section 5.

In practice, the remuneration is implemented by having each key block contain a single coinbase transaction that mints new coins and deposits the funds to the current and previous leaders. As in Bitcoin, this transaction can only be spent after a maturity period of 100 key blocks, to avoid non-mergeable transactions following a fork.

4.5 Microblock Fork Prevention

Since microblocks do not require mining, they can be generated cheaply and quickly by the leader, allowing it to split the brain of the system, publishing different replicated-state-machine states to different machines. This allows for double spending attacks, where different nodes believe the same coins were spent with different transactions.

To demotivate such behavior, we use a dedicated ledger entry that invalidates the revenue of fraudulent leaders. Past work has used such entries in different contexts [22, 4, 13]. In Bitcoin-NG, the entry is called a *poison transaction*, and it contains the header of the first block in the pruned branch as a *proof of fraud*. The poison transaction has to be placed on the blockchain within the maturity window of the misbehaving leader's key block, and before the revenue is spent by the malicious leader. Besides invalidating the compensation sent to the leader that generated the fork, a poison transaction grants the current leader a fraction of that compensation, e.g., 5%. The choice of this value is explained in Section 5.

Only one poison transaction can be placed per cheater, even if the cheater creates many forks. The cheater's revenue that is not relayed to the poisoner is lost.

5 Security Analysis

5.1 Incentives

This section describes how miners with capacity smaller than $1/4$ of the total network are incentivized to follow the protocol. Specifically, miners are motivated to (1) include transactions in their microblocks, (2) extend the heaviest chain, and (3) extend the longest chain. Unlike in Bitcoin, the latter two points are not identical.

Heaviest Chain Extension The motivation for extending the heaviest chain is the same as in Bitcoin. Since the honest majority will extend the heaviest chain, it will remain the main chain with high probability. A dishonest majority may arbitrarily switch to any branch and win [32]. A minority choosing to mine on another branch will not catch up with an honest majority, therefore it will mine on the main chain to ensure its revenues. We therefore argue that the guarantees of Bitcoin-NG are similar to those of Bitcoin [40] with respect to the Termination and Agreement properties of Nakamoto consensus.

Microblocks carry no weight, not even as a secondary index. If they did, it would increase the system's vulnerability to selfish mining [24, 44, 49]. In selfish mining, an attacker withholds blocks it has mined and publishes them judiciously to obtain a superior presence in the main chain. If microblocks carried weight, an attacker could keep secret microblocks and gain advantage by mining on microblocks unpublished to anyone else.

We conclude that Bitcoin-NG does not introduce a new vulnerability to selfish mining strategies, and so Bitcoin-NG is resilient to selfish mining against attackers with less than $1/4$ of the mining power. We therefore argue that the guarantees of Bitcoin-NG are similar to those of Bitcoin with respect to the validity property of Nakamoto consensus.

Transaction Inclusion A leader earns 40% of a transaction's revenue by placing it in a microblock. However, he could potentially improve his revenue by secretly trying to earn 100% of the fee. To do so, first, the leader creates a microblock with the transaction, but does not publish it. Then, he tries to mine on top of this secret microblock, while other miners mine on older microblocks. If the leader succeeds in mining the subsequent key block, he obtains 100% of the transaction fees. Otherwise, he waits until the transaction is placed in a microblock by another miner and tries to mine on top of it.

Consider a miner whose mining power ratio out of all mining power in the system is α. Denote by r_{leader} the revenue of the leader from a transaction, leaving $(1 - r_{\text{leader}})$ for the next miner. In Bitcoin-NG, we have $r_{\text{leader}} = 40\%$. The value of r_{leader} has to be such that the average revenue of a miner trying the above attack is smaller than his revenue placing the transaction in a public microblock as it should:

$$\overbrace{\alpha \times 100\%}^{\text{Win 100\%}} + \overbrace{(1-\alpha) \times \alpha \times (100\% - r_{\text{leader}})}^{\text{Lose 100\%, but mine after txn}} < r_{\text{leader}} ,$$

therefore $r_{\text{leader}} > 1 - \frac{1-\alpha}{1+\alpha-\alpha^2}$. Assuming the power of an attacker is bounded by $1/4$ of the mining power, we obtain $r_{\text{leader}} > 37\%$, hence $r_{\text{leader}} = 40\%$ is within range.

Longest Chain Extension To increase his revenue from a transaction, a miner could avoid the transaction's microblock and mine on a previous block. Then he would place the transaction in its own microblock and try mining the subsequent key block. His revenue in this case must be smaller than his revenue by mining on the transaction's microblock as prescribed:

$$\overbrace{r_{\text{leader}}}^{\substack{\text{Place in}\\ \text{microblock}}} + \overbrace{\alpha(100\% - r_{\text{leader}})}^{\substack{\text{Mine next}\\ \text{key block}}} < \overbrace{100\% - r_{\text{leader}}}^{\substack{\text{Mine on existing}\\ \text{microblock}}} ,$$

therefore $r_{\text{leader}} < \frac{1-\alpha}{2-\alpha}$. Assuming the power of an attacker is bounded by $1/4$ of the mining power, we obtain $r_{\text{leader}} < 43\%$, hence $r_{\text{leader}} = 40\%$ is within range.

Optimal Network Assumption Incentive compatibility cannot be maintained in Bitcoin-NG for an attacker larger than about 29%. For larger attackers, the intersection of the two conditions is empty. But this limit does not come into play in the general case, where Bitcoin-NG, like Nakamoto's blockchain with random tie breaking [25], are secure only against attackers smaller than 23.2% [49] due to selfish mining attacks.

However, under optimal network assumptions, Bitcoin's blockchain is more resilient than Bitcoin-NG: Assuming a zero latency network where an attacker cannot rush messages — i.e., receive a message and send its own such that other nodes receive the attacker's message before the original one — Bitcoin is believed to be secure against selfish mining attackers of size up to almost $1/3$.

Bypassing Fee Distribution We note that a user can circumvent the $40-60\%$ transaction fee distribution by paying no transaction fee, and instead paying the current leader directly, using the coinbase address of the leader's key block. However, a user does not gain a significant advantage by doing so. As we have seen above, paying only the current leader increases the direct motivation of the current leader to place the transaction in a microblock, but reduces the motivation of future miners to mine on this microblock. Moreover, if the leader does not include the transaction before the end of its epoch, subsequent leaders will have no motivation to place the transaction.

Other motives for fee manipulation, such as paying a large fee to encourage miners to choose a certain branch after a fork, apply to Bitcoin as well as Bitcoin-NG, and are outside the scope of this work.

5.2 Other concerns

Wallet Security The possibility of placing a poison transaction allows an attacker that obtains a leader's private key to revoke his revenue retroactively and earn a small amount. However, such an attacker is better off trying to steal the full leader's revenue when it becomes available, therefore the introduction of the poison transaction does not add a significant vulnerability.

Censorship Resistance A central goal of Bitcoin is to prevent a malicious discriminating miner from dropping a user's transactions. Censorship resistance is not impacted by the frequent microblocks of Bitcoin-NG.

First, we note that a leader's absolute power is limited to his epoch of leadership. A malicious leader can perform a DoS attack by placing no transactions in microblocks. Similarly, a benign leader that crashes during his epoch of leadership will publish no microblocks. Their influence ends once the next leader publishes his key block. The impact of such behaviors is therefore similar to that in Bitcoin, where nodes may mine empty blocks, but rarely do.

Assuming an honest majority and no backlog, a user will have her transaction placed in the first block generated by an honest miner. Since at least $3/4$ of the blocks are generated by honest miners, the user will have to wait for $4/3$ blocks on average, or 13.33 minutes. Key block intervals can be set to a rate that would reduce censorship to the minimum allowed by the network without incurring prohibitive deterioration of other metrics.

Resilience to Mining Power Variation Following Bitcoin's success, hundreds of alternative currencies were created [57], most with Bitcoin's exact blockchain structure, and many with the same proof-of-work mechanism. To maintain a stable rate of blocks, different instances of the blockchain tune their proof of work difficulty at different rates: Bitcoin once every 2016 blocks – about 2 weeks, Litecoin [37] every 2016 blocks (produced at a higher rate) – about 3.5 days, and Ethereum [56] on every block – about 12 seconds. However, whichever adjustment rate is chosen, these protocols are all sensitive to sudden mining power drops. Such drops happen when miners are incentivized to stop mining due to a drop in the currency's exchange rate, or to mine for a different currency that becomes more profitable due to a change in mining difficulty or exchange rate of either currency. Such changes are especially problematic for small altcoins. When their value rises, they observe a rapid rise in mining power, and subsequently a drop in mining power once the difficulty rises. Then, since the difficulty is high, the remaining miners need a longer time to generate the next block, potentially orders of magnitude longer.

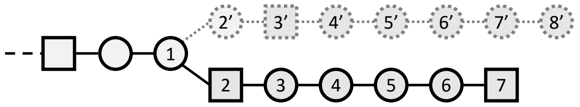


Figure 3: Key block fork. Blocks *B* and *C* have the same chain weight, and the fork is not resolved until key block *D* is published.

In Bitcoin-NG, difficulty adjustments can create a similar problem; however, it only affects key blocks. Microblocks are generated at the same constant rate. As a consequence, in case of a sudden mining power drop, Bitcoin-NG's censorship resistance is reduced, as key blocks are generated infrequently. If a malicious miner becomes a leader, it will generate microblocks until an honest leader finds a key block. Nevertheless, transaction processing continues at the same rate, in microblocks. Additionally, even until the difficulty is tuned to a correct value, the ratio of time during which malicious miners are leaders remains proportional to their mining power.

Forks When issuing microblocks at a high frequency, Bitcoin-NG observes a fork almost on every key block generation, as the previous leader keeps generating microblocks until it receives the key block (Figure 2). These forks are resolved quickly — once the new key block arrives at a node, it switches to the new leader. In comparison, when running Bitcoin at such high frequency, forks are only resolved by the heaviest chain extension rule, and since different miners may mine on different branches, branches remain extant for a longer time compared to Bitcoin-NG.

Bitcoin-NG may also experience key block forks, where multiple key blocks are generated after the same prefix of key blocks, as shown in Figure 3. This rarely happens, due to the low frequency and quick propagation of the small key blocks. The duration of such a fork may be long, lasting until the next key block. The result is therefore infrequent, but long, key block forks.

Although such long forks are undesirable, they are not dangerous. The knowledge of the fork is propagated through the network, and once it reaches the nodes, they are aware of the undetermined state. All transactions that appear only on one branch are therefore uncertain until one branch gains a lead.

Double Spending Double-spending attacks remain a vulnerability in Bitcoin-NG, though to a lesser extent than in Bitcoin.

Consider a Nakamoto blockchain and a Bitcoin-NG blockchain with the same bandwidth, where the Nakamoto block interval is the same as the key-block interval. A double-spending attacker publishes a transaction t_A, receives a service from a merchant, and publishes an alternative conflicting transaction t_B. A merchant that requires very high confidence should wait for several Nakamoto blocks, or an equivalent number of Bitcoin-NG key blocks. With lower confidence requirements, the guarantees of the protocols differ.

In Nakamoto's blockchain, blocks are infrequent, and transactions are collected by miners until they find a block. Until that time, a transaction t_A can be replaced by another transaction t_B without cost. Publication of conflicting transactions with different destinations is prohibited by the standard Bitcoin software, which also warns the user of conflicting transactions propagating in the network [30].

In contrast, in Bitcoin-NG, microblocks are frequent, and so a leader commits to a transaction by placing it in a microblock. It cannot place t_B without forming a fork and subsequently losing all of its prize from its leadership epoch via a poison transaction.

Other attacks are still possible, where a miner mines before the microblock of transaction t_A and later places a conflicting t_B. Here, the attacker loses the fees of all transactions in pruned microblocks, but this may be worthwhile since the loot from the double-spend can be arbitrarily high. An attacker can mine to prune the chain in advance, and then place a conflicting transaction, or try to prune after the fact.

Reasoning about such attacks calls for a formalization of the attacker's incentives and power. We defer formal analysis that quantifies the security guarantees of Bitcoin-NG and Nakamoto's blockchain to future work. In practice, merchants perform risk analysis to choose a strategy appropriate for their business.

6 Metrics

We now detail novel metrics by which blockchains can be evaluated. These metrics are designed to evaluate the unique properties of Nakamoto consensus.

Consensus Delay Intuitively, *consensus delay* is the time it takes for a system to reach agreement. We start by defining, for a specific execution and time, how long back nodes have to look to find a point where they agree on the state.

In a specific execution of an algorithm, given a time t and a ratio $0 < \varepsilon \leq 1$, the ε *point consensus delay* is the smallest time difference Δ such that at least $\varepsilon \cdot |\mathcal{N}|$ of the nodes at time t report the same state machine transition prefix up to time $t - \Delta$. An example for the Bitcoin protocol is illustrated in Figure 4.

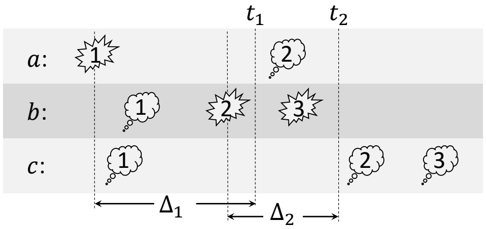


Figure 4: Point-consensus delay example with three Bitcoin nodes a, b, and c that generate blocks at heights 1, 2, and 3 (explosions) and learn that these blocks are in the main chain (clouds). Intervals Δ_1 and Δ_2 are the 50%-point consensus delays at times t_1 and t_2, respectively: At least a majority of the nodes at t_i agree on the history until $t_i - \Delta_i$.

The consensus delay is the best point-consensus-delay the system achieves for a certain fraction of the time, on average. More formally, the (ε, δ) *consensus delay* of a system is the δ-percentile ε-point-consensus-delay. For example, if 90% of the time, 50% of the nodes agree on the state of the state machine 10 seconds ago (but not less than that), then the $(50\%, 90\%)$-consensus delay is 10 seconds.

Fairness We calculate two ratios: (1) the ratio of transitions not coming from the largest miner with respect to all transitions, and (2) the ratio of mining power not owned by the largest miner with respect to all mining power. We call the ratio of these ratios the *fairness*.

Optimally the fairness is 1.0: The largest miner and the non-largest miners' representation in the transitions set should be the same as their respective mining powers.

Mining Power Utilization The security of a proof-of-work system derives from the mining power used to secure it; that is, the mining power an attacker has to outrun to obtain disproportionate control. The *mining power utilization* is the ratio between the mining power that secures the system and the total mining power. Mining power wasted on work that does not appear on the blockchain accounts for the difference.

Subjective Time to Prune Due to the probabilistic nature of Nakamoto consensus, a node may learn of a state machine transition and subsequently learn that this transition has not occurred – that it was pruned from history. This is the case with pruned branches in Bitcoin.

The δ time to prune is the δ-percentile of the difference between the time a node learns about a transition that will eventually be pruned, and the time it learns that this transition has not occurred. This implies what time a user has to wait to be confident a transition has occurred. Note that this metric only considers transitions that are eventually pruned. Figure 5 illustrates an example with the Nakamoto Blockchain.

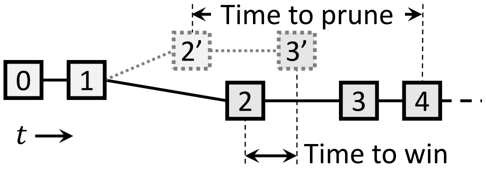


Figure 5: A fork in the blockchain with blocks drawn at their generation times, on a time X axis. *Subjective time to prune* is measured from when a node learns of a block in a branch until it realizes what the main chain is. *Time to win* is measured from the creation time of a block until the last time a node generated a conflicting block.

Time to Win The δ time to win is the δ percentile of the difference between the first time a node believes a never-to-be-pruned-transition has occurred and the last time a (different) node disagrees, believing an alternative transition has occurred. It is zero if there are no disagreements, or if the latter time is earlier. Figure 5 illustrates an example for the Bitcoin protocol.

7 Experimental Setup

We evaluate Bitcoin and Bitcoin-NG with 1000-node experiments running in real time on an emulated network.

Implementation For Bitcoin we run the standard client (release 0.10.0), hereinafter *Bitcoin*, with minimal instrumentation to log sufficient information.

We implemented all Bitcoin-NG elements that are significant for a performance analysis in the absence of an adversary, by modifying the standard Bitcoin client (release 0.10.0). We did not implement the fee distribution and the microblock signature check. Both elements have negligible impact on performance — fee distribution requires about one fixed point operation per transaction and signature checking adds several milliseconds per microblock.

Simulated Mining The time it takes a miner to find a solution follows a geometric probability distribution, which can be approximated as an exponential distribution due to the improbability of a success in each guess and the rate of guessing.

In our experiments we replace the proof of work mechanism with a scheduler that triggers block generation at different miners with exponentially distributed intervals.

Mining Power The probability of mining a block is proportional on average to the mining power used for solving the cryptopuzzle. Since blocks are generated at average set intervals and the total amount of mining power is large, the interval between block generation events of a small miner is extremely large. A single home miner using dedicated hardware is unlikely to mine a block for years [54].

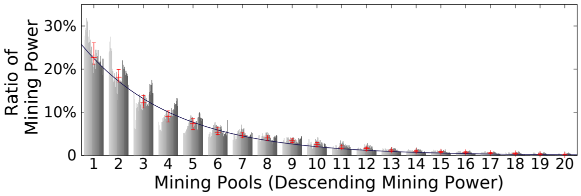


Figure 6: Error bars represent the 75th, 50th and 25th percentiles of the corresponding batch.

Consequently, mining power tends to centralize in the form of industrial mining and open mining pools. Industrial miners are companies that operate large-scale mining facilities. Smaller miners that run private mining rigs typically join forces and form mining *pools*. All members of a pool work together to mine each block, and share their revenues when one of them successfully mines a block.

To reflect in our setup the varying power of miners, we examined the hash power distribution among Bitcoin mining entities. The information we require for the analysis, the identity of the entities generating each block, is voluntarily provided by miners. We used a public API [10] to gather this information for the year ending on August 31, 2015. We note that about 9% of the blocks are unidentified. We considered each such block as generated by a different individual miner.

For each week of the year, we calculate the *weekly mining power* of each entity, and assign rank 1 to the largest weekly mining power, rank 2 to the second largest, and so on. Figure 6 shows the weekly mining power of each entity by rank up to 20. Bars of the same shade at different ranks show the distribution of a specific week. Each batch of bars represents the collection of ratios for the n^{th} highest block generating pool. We note that the ranks of different entities is not preserved throughout the weeks. The y-axis represents the weekly ratio of blocks generated by a pool.

To model the size distribution of mining entities, we approximate it with an exponential distribution with an exponent of -0.27. It yields a 0.99 coefficient of determination compared with the medians of each rank.

Network The structure of Bitcoin's overlay network is complicated, and much of it is intentionally hidden to preserve Bitcoin's security against denial of service (DoS) and to maintain participants' privacy. (Other work [29, 41] discusses details on the peer-to-peer network.) Nodes do not reveal their neighbors, but provide superset of nodes they have discovered. Many of the nodes are hidden behind firewalls making it difficult to even estimate the full size of the network. The latency among nodes is unknown. Moreover, for many of the metrics that we measure, a critical measure is the time it

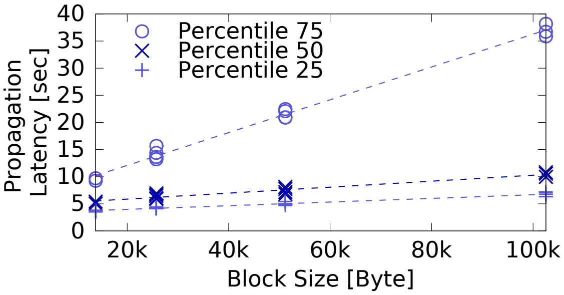


Figure 7: In our system, block propagation time grows linearly with block size. This qualitatively matches the linear relation observed in measurements of the operational Bitcoin network [18].

takes between the generation of a block by some miner and the time at which another miner starts mining on it. The block not only has to be propagated and verified by the second miner, but that second miner must also propagate the details to its mining hardware. In the case of mining pools with many distant worker miners, this may incur a non-negligible delay.

Lacking an existing model of the system, we construct a random network by connecting each node to at least 5 other nodes, chosen uniformly at random. We measured the latency to all visible Bitcoin nodes from a single vantage point on April 7th, 2015, and created a latency histogram. We then set the latency among each pair of nodes in the experiments based on this histogram. The bandwidth is set to about 100kbit/sec among each pair of nodes.

To verify the validity of our setup and topology, we compare Bitcoin's propagation properties in our setup and in the operational system. We perform experiments with different block sizes while changing the block frequency so that the transaction-per-second load is constant. Figure 7 shows a linear relation between the block size and the propagation time, similar to the linear relation measured in the Bitcoin operational network by Decker and Wattenhofer [18].

No Transaction Propagation The goal of this work is to optimize the consensus mechanism of the Blockchain. However, when generating blocks at high frequencies, the overhead of filling in the blocks by generating and propagating transactions becomes a dominant factor with Bitcoin's current implementation. This is not an inherent property of Bitcoin's protocol, or of a Blockchain protocol in general. To reduce the noise caused by the transaction generation and propagation mechanism, we reduce transaction handling to the minimum. Before starting an experiment, we initialize the blockchain with artificial transactions and top up the mempools (the data structure storing yet-to-be-serialized transactions) of all nodes with the same set of transactions. The transactions are of identical size; the operational Bitcoin system as of today, at 1MB blocks every 10 minutes, has a bandwidth of 3.5 such transactions per second.

8 Evaluation

We evaluate Bitcoin-NG and compare it with Bitcoin in two sets of experiments, varying block frequency and block size.

Overall, the experiments show that it is possible to improve Bitcoin's consensus delay and bandwidth by tuning its parameters, but its performance deteriorates dangerously on all security-related metrics. Bitcoin-NG qualitatively outperforms Bitcoin, as it suffers no such deterioration, while enjoying superior performance in almost all metrics across the entire measured range. The bandwidth of Bitcoin-NG is only limited by the processing speed of the individual nodes, as higher throughput does not introduce key-block forks. The consensus delay is determined directly by the network propagation time, because in the common case all nodes agree on the main chain once they receive the latest key block.

In the experiments that follow, we choose the 90th percentile. Lower percentiles maintain the same trends, and very low percentiles show excellent performance – there is always a small subset of nodes that has the correct chain. However, with higher percentiles, the results are lost in the noise. With 1000 nodes and at high percentiles, e.g., 99%, we are measuring the 10th slowest node. Since there are always a few nodes that lag behind, either consistently or temporarily, the results then are dominated by this random behavior, and the trends are not visible.

We measure the metrics we introduced by instantiating them to Nakamoto's blockchain and to Bitcoin-NG as follows.

Consensus delay We take the $(90\%, 90\%)$-consensus delay based on block generation times. Point-consensus-delay for Bitcoin is illustrated in Figure 4. As mentioned in Section 5, a user who requires high confidence (e.g., 99%) will not gain better latency with Bitcoin-NG, and must wait for several key blocks to accept a transaction as completed. The guarantees in such cases are similar to those of Bitcoin with the same block interval as Bitcoin-NG's key-block interval.

Fairness We calculate the proportion of (1) the ratio of blocks in the main chain not generated by the largest miner with respect to all blocks in the main chain, and (2) the ratio of blocks not generated by the largest miner with respect to all generated blocks.

Mining power utilization We calculate the proportion between the aggregate work of the main chain

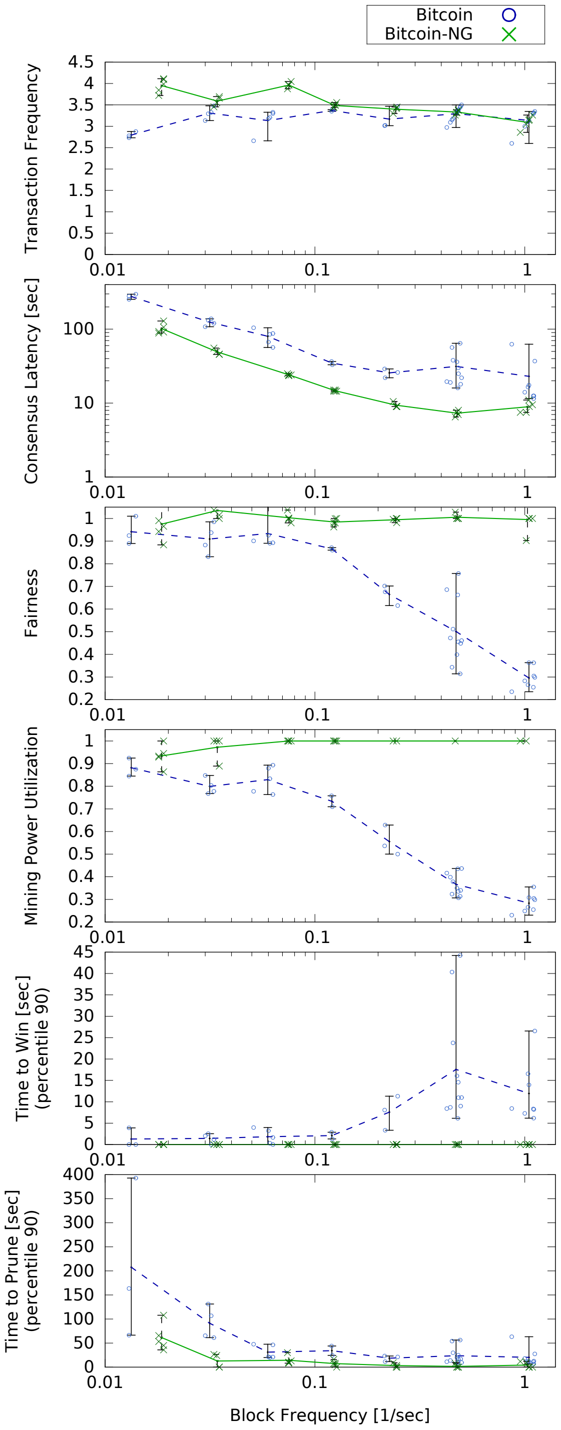


Figure 8: Reducing latency.

blocks and all blocks. In Bitcoin-NG, difficulty is only accrued in key blocks, so microblock forks do not reduce mining power utilization.

Time to prune For each node and for each branch, we measure the time it took for the node to prune this branch. This is the time between the receipt of the first branch block and the receipt of the main chain block that is longer than this branch (Figure 5). We take the 90th percentile of all samples.

Time to win We take the 90th percentile of the time from the generation of each main-chain block to the last time another miner generates a block that is not its descendant (Figure 5).

Experiments We run multiple experiments with different parameters. The figures show the average value for each group of measurements with error bars marking the extreme values. The sampled values are shown as markers.

For each execution we run for 50-100 Bitcoin blocks or Bitcoin-NG microblocks. We perform multiple short runs since all transactions are preloaded for each execution. The mean key-block interval in our experiments is 10 seconds, so each experiment includes leader changes. We do not consider cases where key-block forks occur, since in reality one would choose a much larger key-block interval, e.g., 10 minutes, making key-block forks extremely rare (more rare than with the operational Bitcoin system).

8.1 Block Frequency

First, we run experiments targeted at improving the consensus delay. For Bitcoin, we vary the frequency of block generation by reducing the proof-of-work difficulty. For Bitcoin-NG, keeping the key block generation at one every 100 seconds, we vary the frequency of microblock generation. For each frequency, we choose the block size (microblock size for Bitcoin-NG) such that the payload throughput is identical to that of Bitcoin's operational system, that is, one 1MB block every 10 minutes. Figure 8 shows the results.

We confirm that the bandwidth, measured as transaction frequency, is close to 3.5, the operational Bitcoin rate of for such transactions. In our experiments, Bitcoin's bandwidth is smaller than that of Bitcoin-NG, giving Bitcoin an advantage with respect to the other metrics.

As expected, a higher block frequency reduces Bitcoin's consensus latency as transactions are placed in the ledger at a higher frequency. Time to prune improves significantly as block frequency increases. Nevertheless, Bitcoin's frequent forks leave it with higher consensus latency and time to prune than Bitcoin-NG. We note that although they can be made arbitrarily rare, key block forks do occur. Such key-block forks are only resolved once one branch has more key blocks than the others, resulting in a long time to prune if key block intervals are long.

Bitcoin's mining power utilization drops quickly as frequency increases, tending towards $1/4$, the size of the largest miner. At the extreme, block generation is so fast that by the time a miner learns of a block generated by another miner, that other miner has generated more blocks. Then, only the largest miner generates main chain blocks, and the other miners catch up. This also implies the deterioration of fairness, as forks are likely to be resolved by the largest miner extending its preferred branch. As miners struggle to catch up with the leading pack, slow miners mine on old blocks and the time to win metric increases.

Since contention in Bitcoin-NG is limited to key block generation, forks remain rare despite high frequencies of microblocks. Increasing the microblock frequency achieves reduction of both consensus delay and time to prune. All other metrics are unaffected and remain at the optimal level.

In the low-frequency experiments of Bitcoin-NG, we observe a slight mining power utilization decrease and time to prune increase. This is an artifact of the experimental setup. We run the experiments over a set number of blocks, therefore these low contention experiments run for an extended period, enough to observe key block forks. Note, however, that a realistic Bitcoin-NG implementation can space the key blocks much further apart without affecting performance. Then, due to their small size, key-block forks are highly unlikely, even more so than with standard blocks of Nakamoto's blockchain at the same rate, due to the small size of the key blocks.

8.2 Block Size

To study bandwidth scalability, we run experiments with different block sizes. We use high frequencies, similar to those of Ethereum [12], setting Bitcoin's block frequency to 1/10sec and Bitcoin-NG's microblock frequency to 1/10sec and key block frequency to 1/100sec. Figure 9 shows the result.

As expected, the transaction frequency increases with block size; the horizontal line shows the operational Bitcoin rate.

Large blocks take longer to verify and propagate. Therefore, although block frequency is constant, the time it takes for a miner to learn of a new block is longer, and so the chance for forks increases.

These experiments demonstrate the expected trade-off between bandwidth and latency. Consensus latency increases due to forks, as it takes longer to choose the main chain. The time to win also increases, as blocks take longer to catch up with the larger blocks, as does time to prune due to the many forks.

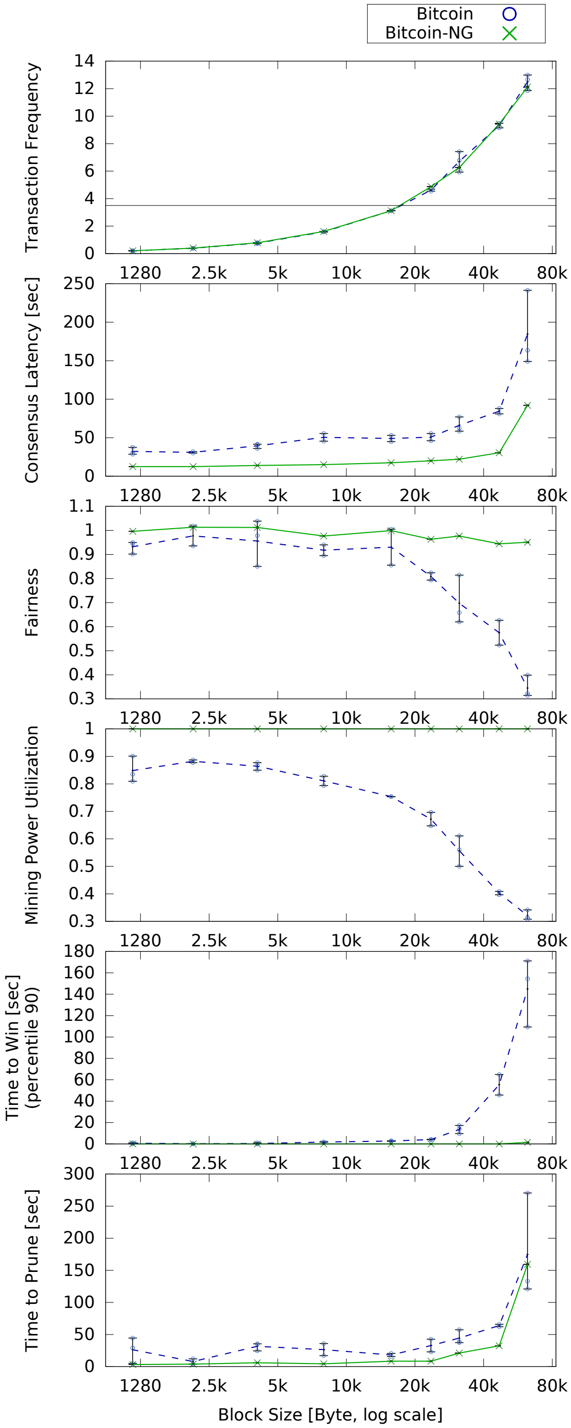


Figure 9: Increasing throughput.

While this trade-off may be acceptable, allowing for some hunt for a sweet spot on the trade-off curve, the real problem pertains to security. The forks cause significant mining power loss, reaching about 80% at Bitcoin's bandwidth (though at a higher block frequency), making the system vulnerable to attackers that are much smaller.

Even more detrimental is the reduction in fairness. Even a minor degradation in fairness is dangerous, since it provides incentives to miners to avoid losses by joining forces to enjoy the advantage of mining in a larger pool. This leads to centralization of the mining power, obviating Bitcoin's security properties.

Bitcoin-NG demonstrates qualitative improvement, suffering no significant degradation in the security-related metrics of fairness and mining power. Under heavy load, however, the clients are approaching their processing capacity, making it hard for them to keep up, and we observe degradation in consensus latency and time to prune.

9 Related Work

Model As in Bitcoin [43] and enhancements thereof [56, 51, 36], the goal of Bitcoin-NG is to implement an RSM in an open system. The exact assumptions and guarantees are explored in different works [11, 40, 26]. Our model is similar to those of Aspnes et al. [2] and Garay et al. [26], and our definition of Nakamoto Consensus is similar to that of Garay et al. [26]. These are different from the model and goal of classical Byzantine fault tolerant RSMs. The latter, by and large, (1) assume static or slow-to-change membership, allowing for quorum systems and reconfigurations thereof, and (2) do not guarantee fairness of representation of honest parties in the state machine transitions.

The problem of leader election was apparently first formulated and solved in 1977 by Gerard LeLann [34]. In 1982, Hector Garcia-Molina addressed the problem in a distributed system that admits failures [27]. Since then leader election has been extensively used to improve the performance of distributed systems (e.g., [20, 42]). In these classical consensus protocols, the leader's role is to propose decisions that have to be confirmed by a quorum. This can be compared to blockchain protocols where the block of a leader (as defined here) is confirmed in retrospect by subsequent blocks of subsequent leaders.

GHOST The GHOST protocol of Sompolinsky et al. [51] improves on Bitcoin's scalability by changing its chain selection rule. While, in Bitcoin, the chain with the most work (accumulated over all chain blocks, based on their proofs of work) is the main chain, with GHOST, at a fork, a node chooses the side whose sub-tree contains more work (accumulated over all sub-tree blocks). The benefit is that the heaviest sub-tree choice takes into account proof of work that does not end up in the main chain. Thus, GHOST improves both fairness and the mining power utilization under high contention.

However, in GHOST, blocks on pruned subtrees only affect the selection rule at the branch point. The Bitcoin-NG protocol maintains a small fork rate at high bandwidth and throughput, allowing for better mining power utilization and fairness. Moreover, to use GHOST in an operational system, a challenge remains. In Bitcoin, at any given time, at least one node knows what the main chain is since it knows all of its blocks. In GHOST, this is not the case, and it is possible that no single node has enough information to determine which is the main chain. Our technical report [23] provides an example.

One solution to finding the true main chain in GHOST is to propagate all blocks, or all block headers [51]. However, this exposes the system to denial-of-service attacks, as a malicious node can overwhelm the network with low-difficulty blocks. There may be heuristics to avoid the security danger; we do not address this question, but have evaluated the system by implementing it, propagating all blocks. Under these conditions, GHOST performs worse than Bitcoin as the overhead of propagating all blocks outweighs the benefits of the chain selection rule. Nevertheless, a practical implementation of GHOST, overcoming remaining challenges, can be used to complement Bitcoin-NG and allow for a higher frequency of key blocks.

Inclusive Blockchains Lewenberg et al. [36] replace the blockchain structure with a directed acyclic graph. There still is a main chain, but its blocks may refer to pruned branches to include their transactions. Analysis demonstrates considerable improvement of fairness and mining power utilization. Bitcoin-NG achieves optimal fairness and mining power utilization. Using Bitcoin-NG with an inclusive blockchain to increase key block frequency may prove problematic: Decommissioned leaders could retroactively introduce transactions and have them included by the current leader. This could allow for DoS and double-spending attacks.

Faster Bitcoin Significant effort by Bitcoin's core developers is put into improving the performance of the Bitcoin client and technical aspects of its protocol. While this work can provide significant improvement and enable better scaling, it does not eliminate the inherent limitation that stems from forks forming at high rates.

Stathakopoulou et al. suggest reducing propagation delay in the Bitcoin network [53]. However, their suggestions imply significant compromises on security. First, they have nodes propagate transaction inventories before they know the actual transactions in each inventory; this allows an attacker to swamp the network at no cost by publishing transaction IDs for non-existent transactions. Second, they form a network by having nodes prefer connections with close neighbors — exactly the opposite of the current security-oriented algorithm.

Improving the efficiency of the client [1, 45, 55] can improve propagation time and reduce the collision window (time before *A* hears *B* found a block). However, the improvement is limited — a processing speed increase of $x\%$ (e.g., $x = 200\%$ with [55]) allows for block size increase of $x\%$ at the same fork rate. Bitcoin-NG provides a qualitative improvement that removes the fork rate dependency on block size or rate.

Corallo [17] has built a centralized fast relay for Bitcoin, parallel to the standard peer-to-peer network. It significantly improves network throughput and latency but increases centralized control and reduces fairness — miners outside the fast relay are at a disadvantage.

Off-chain solutions An alternative to improving the bandwidth and latency of the blockchain is to perform transactions off the chain. This basic premise apparently originated in Hearn and Spilman's two-point channel protocol [28]. The Lightning network [48] and duplex micropayment channels [19] allow for payment networks layered on top of a blockchain. The security and privacy guarantees of such payment networks differ from those of Bitcoin; as an extreme example, if the nodes performing transactions over a channel crash, all their transactions are lost, as they were never stored in the blockchain. Moreover, the efficacy of such solutions depends on properties of the emergent payment network, its topology, the amount of value locked in payment channels, as well as the protocol's ability to discover and use payment paths. Overall, these solutions may be suitable for targeted use cases where the additional layer may reduce the number of transactions seen at the lower layers, but, unlike Bitcoin-NG, they do not address the fundamental problem of scaling a Nakamoto-consensus RSM.

Another proposition for improving performance is that of federated chains, known as side chains [4], where transactions can move coins from one chain to another. Sidechains provide extensibility, as different chains can offer different features. However, their contribution for efficiency is limited, as they incur high latencies for crossing chains; moreover, when the payor has funds on one sidechain and the payee would like to spend them on another, the funds have to cross the main chain in order to get the value to their intended destination.

Analysis Given a cryptopuzzle difficulty and a topology, Sompolinsky et al. [52] calculate upper and lower bounds for the growth rate of the Bitcoin main chain. This analysis can be translated to the expected forking frequency at different difficulty levels when there are exactly two miners. Our experiments target a larger number of miners, modeled according to Bitcoin's operational system, that tune difficulty arbitrarily to reach a target main chain extension rate.

Miller and Jansen [39] describe a methodology for evaluating a large-scale Bitcoin blockchain system on a single machine using an event-driven simulator. To facilitate manageable experiment times, they replace time-consuming cryptographic operations with a delay of an appropriate length. In our experiments, we run the original operational client directly on the operating system, emulating only the network properties and mining events.

Incentives Incentive compatibility has been a key issue in the investigation of cryptocurrencies. Babaioff et al. [3] suggest a mechanism to motivate transaction propagation. Lewenberg et al. [35] propose an alternative to the chain structure to motivate the participation of badly-connected miners. Eyal [21] shows that a natural incentive system deters the formation of large open mining pools.

10 Conclusion

As Bitcoin and related cryptocurrencies have become surprisingly popular, they have hit scalability limits. The technical debate to improve scalability has been hampered by a perceived inherent trade-off between performance metrics and security goals of the system. Consequently, the discussions have become acrimonious, long-term solutions have seemed elusive, and the current sentiment has centered around short-term, incremental, compromise solutions.

Bitcoin-NG shows that it is possible to improve the scalability of blockchain protocols to the point where the consensus latency is limited solely by the network diameter and the throughput bottleneck lies only in node processing power. Such scaling is key in allowing for blockchain technology to fulfill its promise of implementing trustless consensus for a variety of demanding applications including payments, digital asset transactions, and smart contracts — at global scale.

Acknowledgments The authors thank Ayush Dubey, Gregory Maxwell, Malte Möser, Weijia Song, the paper's shepherd Jay Lorch, and the anonymous reviewers for their comments on earlier versions of this manuscript.

References

[1] Andresen, G. O(1) block propagation. https://gist.github.com/gavinandresen/#file-blockpropagation-md, retrieved July. 2015.

[2] Aspnes, J. Randomized protocols for asynchronous consensus. *Distributed Computing 16*, 2-3 (2003), 165–175.

[3] Babaioff, M., Dobzinski, S., Oren, S., and Zohar, A. On Bitcoin and red balloons. In *ACM Conference on Electronic Commerce* (Valencia, Spain, 2012), pp. 56–73.

[4] Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra, A., Timn, J., and Wuille, P. Enabling blockchain innovations with pegged sidechains. http://cs.umd.edu/projects/coinscope/coinscope.pdf, 2014.

[5] Bamert, T., Decker, C., Elsen, L., Wattenhofer, R., and Welten, S. Have a snack, pay with Bitcoins. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on* (2013), IEEE, pp. 1–5.

[6] Bellare, M., and Rogaway, P. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security* (1993), ACM, pp. 62–73.

[7] Bitcoin community. Bitcoin source. https://github.com/bitcoin/bitcoin, retrieved Mar. 2015.

[8] Bitcoin community. Protocol rules. https://en.bitcoin.it/wiki/Protocol_rules, retrieved Sep. 2013.

[9] Bitcoin community. Protocol specification. https://en.bitcoin.it/wiki/Protocol_specification, retrieved Sep. 2013.

[10] BlockTrail. BlockTrail API. https://www.blocktrail.com/api/docs#api_data, retrieved Sep. 2015.

[11] Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J. A., and Felten, E. W. Research perspectives on Bitcoin and second-generation cryptocurrencies. In *Symposium on Security and Privacy* (San Jose, CA, USA, 2015), IEEE.

[12] Buterin, V. A next generation smart contract & decentralized application platform. https://www.ethereum.org/pdfs/EthereumWhitePaper.pdf/, retrieved Feb. 2015, 2013.

[13] Buterin, V. Slasher: A punitive proof-of-stake algorithm. https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm/, January 2015.

[14] CNNMoney Staff. The Ashley Madison hack...in 2 minutes. http://money.cnn.com/2015/08/24/technology/ashley-madison-hack-in-2-minutes/, retrieved Sep. 2015.

[15] CoinDesk. Bitcoin venture capital. http://www.coindesk.com/bitcoin-venture-capital/, retrieved Sep. 2015.

[16] Colored Coins Project. Colored Coins. http://coloredcoins.org/, retrieved Sep. 2015.

[17] Corallo, M. High-speed Bitcoin relay network. http://sourceforge.net/p/bitcoin/mailman/message/31604935/, November 2013.

[18] Decker, C., and Wattenhofer, R. Information propagation in the Bitcoin network. In *IEEE P2P* (Trento, Italy, 2013).

[19] Decker, C., and Wattenhofer, R. A fast and scalable payment network with Bitcoin Duplex Micropayment Channels. In *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings* (2015), Springer, pp. 3–18.

[20] Dwork, C., Lynch, N. A., and Stockmeyer, L. J. Consensus in the presence of partial synchrony. *J. ACM 35*, 2 (1988), 288–323.

[21] Eyal, I. The miner's dilemma. In *IEEE Symposium on Security and Privacy* (2015), pp. 89–103.

[22] Eyal, I., Birman, K., and van Renesse, R. Cache serializability: Reducing inconsistency in edge transactions. In *35th IEEE International Conference on Distributed Computing Systems* (2015), pp. 686–695.

[23] Eyal, I., Gencer, A. E., Sirer, E. G., and van Renesse, R. Bitcoin-ng: A scalable blockchain protocol. *arXiv preprint arXiv:1510.02037* (2015).

[24] Eyal, I., and Sirer, E. G. Bitcoin is broken. http://hackingdistributed.com/2013/11/04/bitcoin-is-broken/, 2013.

[25] Eyal, I., and Sirer, E. G. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security* (2014).

[26] Garay, J. A., Kiayias, A., and Leonardos, N. The Bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2015), pp. 281–310.

[27] Garcia-Molina, H. Elections in a distributed computing system. *Computers, IEEE Transactions on 100*, 1 (1982), 48–59.

[28] Hearn, M., and Spilman, J. Rapidly-adjusted (micro)payments to a pre-determined party. https://en.bitcoin.it/wiki/Contract, retrieved Sep. 2015.

[29] Heilman, E., Kendler, A., Zohar, A., and Goldberg, S. Eclipse attacks on Bitcoin's peer-to-peer network. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* (2015), pp. 129–144.

[30] Karame, G. O., Androulaki, E., and Capkun, S. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012), CCS '12, ACM, pp. 906–917.

[31] Kosba, A., Miller, A., Shi, E., Wen, Z., and Papamanthou, C. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. Cryptology ePrint Archive, Report 2015/675, 2015. http://eprint.iacr.org/.

[32] Kroll, J. A., Davey, I. C., and Felten, E. W. The economics of Bitcoin mining or, Bitcoin in the presence of adversaries. In *Workshop on the Economics of Information Security* (2013).

[33] Lamport, L. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems 6*, 2 (Apr. 1984), 254–280.

[34] Le Lann, G. Distributed systems-towards a formal approach. In *IFIP Congress* (1977), vol. 7, Toronto, pp. 155–160.

[35] Lewenberg, Y., Bachrach, Y., Sompolinsky, Y., Zohar, A., and Rosenschein, J. S. Bitcoin mining pools: A cooperative game theoretic analysis. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems* (2015), International Foundation for Autonomous Agents and Multiagent Systems, pp. 919–927.

[36] Lewenberg, Y., Sompolinsky, Y., and Zohar, A. Inclusive block chain protocols. In *Financial Cryptography* (Puerto Rico, 2015).

[37] Litecoin Project. Litecoin, open source P2P digital currency. https://litecoin.org, retrieved Nov. 2014.

[38] Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G. M., and Savage, S. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013* (2013), pp. 127–140.

[39] Miller, A., and Jansen, R. Shadow-Bitcoin: Scalable simulation via direct execution of multi-threaded applications. *IACR Cryptology ePrint Archive 2015* (2015), 469.

[40] Miller, A., and Jr., L. J. J. Anonymous Byzantine consensus from moderately-hard puzzles: A model for Bitcoin. https://socrates1024.s3.amazonaws.com/consensus.pdf, 2009.

[41] Miller, A., Litton, J., Pachulski, A., Gupta, N., Levin, D., Spring, N., and Bhattacharjee, B. Preprint: Discovering Bitcoins public topology and influential nodes. http://cs.umd.edu/projects/coinscope/coinscope.pdf, 2015.

[42] Moraru, I., Andersen, D. G., and Kaminsky, M. Egalitarian Paxos. In *ACM Symposium on Operating Systems Principles* (2012).

[43] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf, 2008.

[44] Nayak, K., Kumar, S., Miller, A., and Shi, E. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. *IACR Cryptology ePrint Archive 2015* (2015), 796.

[45] Pazmiño, J. E., and da Silva Rodrigues, C. K. Simply dividing a Bitcoin network node may reduce transaction verification time. *The SIJ Transactions on Computer Networks and Communication Engineering (CNCE) 3*, 2 (February 2015), 17–21.

[46] Pease, M. C., Shostak, R. E., and Lamport, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (1980), 228–234.

[47] Peck, M. E. Adam Back says the Bitcoin fork is a coup. http://spectrum.ieee.org/tech-talk/computing/networks/the-bitcoin-for-is-a-coup, Aug 2015.

[48] Poon, J., and Dryja, T. The Bitcoin Lightning Network. http://lightning.network/lightning-network.pdf, February 2015. Draft 0.5.

[49] Sapirshtein, A., Sompolinsky, Y., and Zohar, A. Optimal selfish mining strategies in Bitcoin. *CoRR abs/1507.06183* (2015).

[50] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22*, 4 (Dec. 1990), 299–319.

[51] Sompolinsky, Y., and Zohar, A. Accelerating Bitcoin's transaction processing. fast money grows on trees, not chains. In *Financial Cryptography* (Puerto Rico, 2015).

[52] Sompolinsky, Y., and Zohar, A. Secure high-rate transaction processing in Bitcoin. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers* (2015), pp. 507–527.

[53] Stathakopoulou, C. A faster Bitcoin network. Tech. rep., ETH, Zürich, January 2015. Semester Thesis, supervised by C. Decker and R. Wattenhofer.

[54] Swanson, E. Bitcoin mining calculator. http://www.alloscomp.com/bitcoin/calculator, retrieved Sep. 2013.

[55] The Bitcoin Community. Release notes, bitcoin 0.12.0. https://github.com/bitcoin/bitcoin/blob/0.12/doc/release-notes.md, Feb 2012.

[56] The Ethereum community. Ethereum white paper. https://github.com/ethereum/wiki/wiki/White-Paper, retrieved July. 2015.

[57] Wikipedia. List of cryptocurrencies. https://en.wikipedia.org/wiki/List_of_cryptocurrencies, retrieved Oct. 2013.

Exploring Cross-Application Cellular Traffic Optimization with Baidu TrafficGuard

Zhenhua Li [1,2], Weiwei Wang [2], Tianyin Xu [3], Xin Zhong [1,2]
Xiang-Yang Li [1,4,5], Yunhao Liu [1], Christo Wilson [6], Ben Y. Zhao [7]

[1] *Tsinghua University* [2] *Baidu Mobile Security* [3] *University of California San Diego*
[4] *University of Science and Technology of China* [5] *Illinois Institute of Technology*
[6] *Northeastern University* [7] *University of California Santa Barbara*

Abstract

As mobile cellular devices and traffic continue their rapid growth, providers are taking larger steps to optimize traffic, with the hopes of improving user experiences while reducing congestion and bandwidth costs. This paper presents the design, deployment, and experiences with Baidu TrafficGuard, a cloud-based mobile proxy that reduces cellular traffic using a network-layer VPN. The VPN connects a client-side proxy to a centralized traffic processing cloud. TrafficGuard works transparently across heterogeneous applications, and effectively reduces cellular traffic by 36% and overage instances by 10.7 times for roughly 10 million Android users in China. We discuss a large-scale cellular traffic analysis effort, how the resulting insights guided the design of TrafficGuard, and our experiences with a variety of traffic optimization techniques over one year of deployment.

1 Introduction

Mobile cellular devices are changing today's Internet landscape. Growth in cellular devices today greatly outpaces that of traditional PCs, and global cellular traffic is growing by double digits annually, to an estimated 15.9 Exabytes in 2018 [4]. This growing traffic demand has led to significant congestion on today's cellular networks, resulting in bandwidth caps and throttling at major wireless providers. The challenges are more dramatic in developing countries, where low-capacity cellular networks often fail to deliver basic quality of service needed for simple applications [40, 41, 44].

While this is a well known problem, only recently have we seen efforts to address it at scale. Google took the unprecedented step of prioritizing mobile-friendly sites in its search algorithm [9]. This will likely spur further efforts to update popular websites for mobile devices. Recent reports estimate that most enterprise webpages are designed for PCs, and only 38% of webpages are mobile-friendly [24]. More recently, Google released details on their Flywheel proxy service for compressing content for the Chrome mobile browser [29].

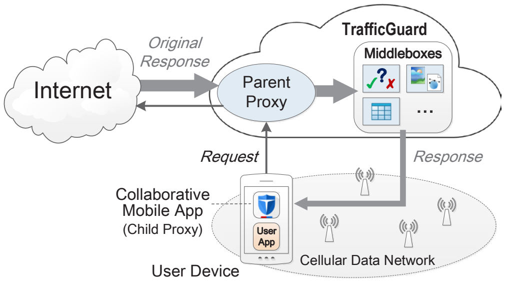


Figure 1: Architectural overview of TrafficGuard.

Competition in today's mobile platforms has led to numerous "walled-gardens," where developers build their own suites of applications that keep users within their ecosystem. The ongoing trend limits the benefits of application-specific proxies, even ones with user bases as large as Google Chrome [29, 18, 25, 21, 42, 43, 36]. In contrast, an alternative approach is to transparently intercept and optimize network traffic across all apps at the OS/network layer. Although some examples of this approach exist [16, 17, 6, 15], little is known about their design or impact on network performance.

This paper describes the design, deployment, and experiences with Baidu TrafficGuard, a third-party cellular traffic proxy widely deployed for Android devices in China [1]. As demonstrated in Figure 1, TrafficGuard is a cloud-based proxy that redirects traffic through a VPN to a client-side mobile app (*http://shoujiweishi.baidu.com*). It currently supports all Android 4.0+ devices, and does not require root privileges. Inside the cloud, a series of software middleboxes are utilized to monitor, filter, and reshape cellular traffic. TrafficGuard was first deployed in early 2014, and its Android app has been installed by

[1] Cellular data usage in Asia differs from that of US/European networks, in that HTTP traffic dominates 80.4% of cellular traffic in China and 74.6% in South Korea [62]. In comparison, HTTPS accounts for more than 50% of cellular traffic in the US [56, 47, 54].

roughly 10 million users. The average number of daily active users is around 0.2 million.

In designing a transparent mobile proxy for cellular traffic optimization, TrafficGuard targets four key goals:

- First, traffic optimization should not harm user experiences. For example, image compression through pixel scaling often distorts webpage and UI (user interface) rendering in user apps. Similarly, traffic processing should not introduce unacceptable delays.
- Second, our techniques must generalize to different apps, and thus proprietary APIs or data formats should be avoided. For example, Flywheel achieves significant traffic savings by transcoding images to the WebP format [27]. Though WebP offers high compression, not all apps support this format.
- Third, we wish to limit client-side resource consumption, in terms of memory, CPU, and battery. Note that the client needs to collaborate well with the cloud using a certain amount of resources.
- Finally, we wish to reduce system complexity, resource consumption, and monetary costs on the cloud side. In particular, the state information maintained for each client should be carefully determined.

In this paper, we document considerations in the design, implementation, and deployment of TrafficGuard. *First*, we analyze aggregate cellular traffic measurements over 110K users to understand the characteristics of cellular traffic in China. This gave us insights on the efficacy and impact of traditional data compression, as well as the role of useless content like broken images in cellular traffic. *Second*, we adopt a lightweight, adaptive approach to image compression, where more considerate compression schemes are constructed to achieve a sweet spot on the image-quality versus file-size tradeoff. This helps us achieve traffic savings comparable to Flywheel (27%) at roughly 10%–12% of the computation overhead. *Third*, we develop a customized VPN tunnel to efficiently filter users' unwanted traffic, including overnight, background, malicious, and advertisement traffic. *Finally*, we implement a cloud-client paired proxy system, and integrate best-of-breed caching techniques for duplicate content detection. The cloud-client paired design allows us to finely tune the tradeoff between traffic optimization and state maintenance.

TrafficGuard is the culmination of these efforts. For installed users, it reduces overall cellular traffic by an average of 36%, and instances of traffic overage (*i.e.,* going beyond the users' allotted data caps) by 10.7 times. Roughly 55% of users saw more than a quarter reduction in traffic, and 20% of users saw their traffic reduced by half. TrafficGuard introduces relatively small latency penalties (median of 53 ms, mean of 282 ms), and has little to no impact on the battery life of user devices.

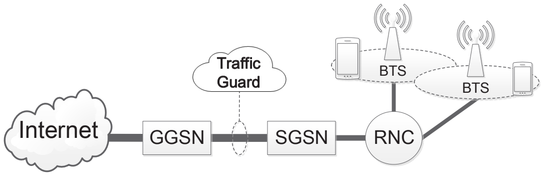


Figure 2: Potential integration of TrafficGuard into a 3G cellular carrier. Integration for 4G would be similar.

While already successful in its current deployment, TrafficGuard can achieve even higher efficiency if cellular carriers (are willing to) integrate it into their infrastructure. As demonstrated in Figure 2, carriers could deploy TrafficGuard between the GGSN (Gateway G-PRS Support Node) and SGSN (Serving GPRS Support Node). Then the optimized traffic is further transferred to the RNC (Radio Network Controller) and BTS (Base Transceiver Station). This would greatly simplify both the cloud-side and client-side components of TrafficGuard, and further reduce latency penalties for users.

Finally, we note that while Baidu does not have an internal IRB (institutional review board [13]) review process, all reasonable steps were taken at Baidu to protect user privacy during this study. All users who participated in the study opted-in as volunteers with informed consent, and full traffic traces were limited to one week of measurements (all other datasets are anonymized logs). Wherever possible, analysis was limited to anonymized metadata only. When necessary, content analysis was done on aggregate data, and fully decoupled from any user identifiers or personally identifiable information.

2 State-of-the-Art Systems

This section briefly surveys state-of-the-art mobile traffic proxy systems. As listed in Table 1, we compare seven systems with TrafficGuard. We focus on five of the most important and ubiquitous features supported by these systems: 1) image compression, 2) text compression, 3) content optimization, 4) traffic filtering, and 5) caching. In each case, we highlight the strengths of different approaches, as well as the shortcomings, which motivated our design of TrafficGuard.

Since most mobile traffic proxy systems are closed-source, we rely on a variety of methods to determine their features. The implementation of Google Flywheel is described in [29]. For Opera Turbo, UCBrowser (proxy), and QQBrowser (proxy), we are able to uncover most of their features through carefully controlled experiments. Specifically, we set up our own web server, used these proxies to browse our own content hosted by the server, and carefully compared the data sent by the server with what was received by our client device. Unfortunately, Opera Max, Microsoft Data Sense, and Onavo Extend

Table 1: Comparison of state-of-the-art mobile traffic proxy systems. "?" means unknown.

System	Image Compression	Text Compression	Content Optimization	Traffic Filtering	Caching
Google Flywheel	Transcoding to WebP	Yes	Lightweight error page	Safe Browsing	Server-side
Opera Turbo	Transcoding to WebP	Yes	Pre-executing JavaScript	Ad blocking	?
UCBrowser	Pixel Scaling	Yes	No	Ad blocking	?
QQBrowser	Transcoding to WebP	Yes	No	Ad blocking	?
Opera Max [17] (China's version)	Transcoding PNG to JPEG	Yes	No	Restricting overnight traffic	?
Microsoft Data Sense	?	Yes	No	Restricting background traffic, and ad blocking	?
Onavo Extend	Transcoding PNG and large GIF to JPEG	Yes	No	No	Client-side
TrafficGuard	Adaptive quality reduction	No	Attempting to discard useless content	Restricting overnight and background traffic, ad blocking, Safe Browsing	Server-side, and VBWC on both sides

use encrypted proxies, and thus we can only discover a subset of their implementation details.

First, we examine the image compression techniques. Three systems transcode images to WebP, which effectively reduces network traffic [29]. However, this only works for user apps that support WebP (*e.g.,* Google Chrome). Similarly, Opera Max and Onavo Extend transcode PNGs to JPEGs, and Onavo Extend also transcodes large GIFs to JPEGs. Taking a different approach, UCBrowser rescales large images ($> 700 \times 700$ pixels) to small images ($< 150 \times 150$ pixels). Although rescaling reduces traffic, it could harm user experiences by significantly degrading image qualities. In contrast to these systems, TrafficGuard uses an adaptive quality reduction approach that is not CPU intensive, reduces traffic across apps, and generally does not harm user experiences (see § 5.1).

Second, we find that all the seven systems compress textual content, typically with gzip. However, our large-scale measurement findings (in § 3.2.2) reveal that the vast majority of textual content downloaded by smartphone users is very short, meaning that compression would be ineffective. Thus, TrafficGuard does not compress texts, since the CPU overhead of decompression is not worth the low (1.36%) HTTP traffic savings.

Third, we explore the *content optimization* strategies employed by mobile traffic proxies. We define *content optimization* as attempts to reduce network traffic by altering the semantics or functionality of content. For example, Flywheel replaces HTTP 404 error pages with a lightweight version. More aggressively, Opera Turbo executes JavaScript objects at the proxy, so that clients do not need to download and execute them. Although this can reduce traffic, it often breaks the original functionality of websites and user apps, *e.g.,* in the controlled experiments we often noticed that JavaScript functions like onscroll() and oninput() were not properly executed by Opera Turbo. Rather than adopt these approaches, TrafficGuard validates HTTP content and attempts to discard useless content like broken images (see § 5.2).

Fourth, we observe that many of the target systems implement traffic filtering. Four systems block advertisements, plus Flywheel using Google Safe Browsing [8] to block malicious content. Opera Max attempts to restrict apps' traffic usage during the night, when users are likely to be asleep. Microsoft Data Sense takes things a step further by also restricting traffic from background apps, under the assumption that apps which are not currently interactive should not be downloading lots of data. We discover that all these filtering techniques are beneficial to users (see § 3.2.4), and thus we incorporate all of them into TrafficGuard (see § 5.3).

Finally, we study the caching strategies of existing systems. Flywheel maintains a server-side cache of recently accessed objects, while Onavo Extend maintains a local cache (of 100 MB by default). In contrast, TrafficGuard adopts server-side strategies by maintaining a cache at the proxy (see § 4.2), as well as implementing Value-based Web Caching (VBWC) between the client and server (see § 5.4). Although we evaluated other sophisticated caching strategies (see Appendix A), we ultimately chose VBWC because it offers excellent performance and is straightforward to implement.

3 Measuring Cellular Traffic

In this section, we present a large-scale measurement study of cellular traffic usage by Android smartphone users. Unlike prior studies [37, 46, 52, 39, 34, 62], our analysis focuses on content and metadata. Using this dataset, we identify several key performance issues and tradeoffs that guide the design of TrafficGuard.

3.1 Dataset Collection

The ultimate goal of TrafficGuard is to improve smartphone users' experiences by decreasing network usage and filtering unwanted content. To achieve this goal, we decided to take a measurement-driven methodology, *i.e.,* we first observed the actual cellular traffic usage patterns of smartphone users, and then used the data to drive our design and implementation decisions.

Table 2: General statistics of our collected TGdataset.

Collection period	03/21 – 03/27, 2014
Unique users	111,910
Total requests	162M
Dataset size	1324 GB (100%)
Non-HTTP traffic (plus TCP/IP)	259 GB (19.6%)
HTTP traffic (plus TCP/IP)	1065 GB (80.4%)
HTTP header traffic	107 GB (8.1%)
HTTP body traffic	875 GB (66.1%)

When we first deployed TrafficGuard between Jan. 5–Mar. 31, 2014, the system only monitored users' cellular traffic; it did not filter or reshape traffic at all. We randomly invited users to test TrafficGuard from $\sim$100M existing mobile users of Baidu. We obtained informed consent from volunteers by prominently informing them that full traces of their cellular traffic would be collected and analyzed. We assigned a unique *ClientToken* to each user device that installed the mobile app of TrafficGuard.

We used two methods to collect packet traces from volunteers. For an HTTP request, the TrafficGuard app would insert the ClientToken into the HTTP header. The TrafficGuard cloud would then record the request, remove the injected header, complete the HTTP request, and store the server's response. However, for non-HTTP requests (most of which are HTTPS), it was not possible for the TrafficGuard cloud to read the injected *ClientToken* (we did not attack secure connections via man-in-the-middle). Thus, the TrafficGuard app locally recorded the non-HTTP traffic, and uploaded it to the cloud in a batch along with the ClientToken once per week. These uploads were restricted to WiFi [2], in order to avoid wasting volunteers' cellular data traffic. In both cases, we also recorded additional metadata like the specific app that initiated each request, and whether that app was working in the foreground or background.

We collected packet traces from volunteers for one week, between Mar. 21–27, 2014. In total, this dataset contains 320M requests from 0.65M unique ClientTokens. However, we observe that many user devices in the dataset only used their cellular connections for short periods of time. These *short-term* users might have good WiFi availability, or might be using their cellular connections but did not (remember to) run the mobile app of TrafficGuard. To avoid bias, we focus on the traces belonging to 111,910 *long-term* users who used their cellular connections in at least four days during the collection period. This final dataset is referred to as TGdataset, whose general statistics are listed in Table 2.

Table 3: Statistics of HTTP content in TGdataset.

Type	% of	% of	Size (KB)	
	Requests	**HTTP Traffic**	**Median**	**Mean**
Image	**32%**	**71%**	**5.7**	**15.5**
Text	**49%**	**15.7%**	**0.2**	**2.2**
Octet-stream	10%	5.5%	0.4	3.8
Zip	8.1%	5.1%	0.5	4.3
Audio & Video	0.03%	2.6%	407	614
Other	0.87%	0.1%	0.3	0.7

3.2 Content Analysis

Below, we analyze the content and metadata contained in TGdataset. In particular, we observe that today's cellular traffic can be effectively optimized in multiple ways.

3.2.1 General Characteristics

We begin by presenting some general characteristics of TGdataset. As listed in Table 2, 80.4% of TGdataset is HTTP traffic, most of which corresponds to the bodies of HTTP messages. This finding is positive for two reasons. First, it means content metadata (*e.g.,* Content-Length and Content-Type) is readily available for us to analyze. Second, it is clear that the TrafficGuard system will be able to analyze and modify the vast majority of cellular traffic, since it is in plaintext.

Table 3 presents information about the types of HTTP content in TGdataset. We observe that images are the second most frequent type of content, but consume 71% of the entire HTTP traffic. Textual content is the most frequent, while non-image binary content accounts for the remainder of HTTP traffic. We manually analyzed many of the octet-streams in our dataset and found that they mainly consist of software and video streams.

3.2.2 Size and Quality of Content

Next, we examine the size and quality of content in TGdataset, and relate these characteristics to the compressibility of content.

Images. Four image types dominate in our dataset: JPEG, WebP, PNG, and GIF. Certainly, all four types of images are already compressed. However, we observe that 40% of images are *large*, which we define as images of $w \times h$ pixels such that $w \times h \geq 250000 \wedge w \geq 150 \wedge h \geq 150$ (refer to § 5.1 for more details of image categorization). Some images even have over 4000×4000 pixels (exceeding 10 MB in size) in extreme cases.

More importantly, we observe that many JPEGs have high *quality factors* (QFs). QF determines the strength of JPEG's lossy-compression algorithm, with $\text{QF} = 100$ causing minimal loss but a larger file size. The median QF of JPEGs in TGdataset is 80 while the average is 74. Such high-quality images are unnecessary for most cellular users, considering their limited data plans and screen sizes. This presents us with an optimization opportunity that TrafficGuard takes advantage of (see § 5.1).

[2] Certainly TrafficGuard also has the *capability* of helping mobile users save WiFi traffic, just like what Google Flywheel does. However, at the moment TrafficGuard only targets at saving cellular traffic for two reasons. First, WiFi users generally do not care about the traffic usage since they do not pay for their Internet access in terms of traffic usage. Second, proxy-based traffic saving inevitably leads to latency penalty and thus would impact WiFi users' experiences.

Table 4: Validity and usefulness of images.

Type	% of Requests	% of Image Traffic	Image Size (KB): Median	Mean
Correct	87%	95.9%	5.4	14.8
Broken	**10.6%**	**3.2%**	**0.13**	**3.2**
Blank	2.3%	0.57%	0	0
Incomplete	0.1%	0.21%	0.01	5.0
Inconsistent	0.04%	0.16%	4.8	33

Textual content. The six most common types of textual content in TGdataset are: JSON, HTML, PLAIN, JavaScript, XML, and CSS. Compared with images, textual content is much smaller: the median size is merely 0.2 KB. Compressing the short texts with the size less than 0.2 KB (*e.g.,* with gzip, bzip2, or 7-zip) cannot decrease their size; in fact, the additional compression metadata may even *increase* the size of such textual data.

Surprisingly, we find that compressing the other, larger half (> 0.2 KB) of textual content with gzip brings limited benefits — it only reduced the HTTP traffic of texts by 8.7%, equal to 1.36% ($= 8.7\% \times 15.7\%$) of total HTTP traffic. Similarly, using bzip2 and 7-zip could not significantly increase the compression rate. However, decompressing texts on user devices does necessitate additional computation and thus causes battery overhead. Given the limited network efficiency gains and the toll on battery life, we opt to not compress texts in TrafficGuard, unlike all other systems as listed in Table 1.

Other content. For the remaining octet-stream, zip, audio & video content, we find that compression provides negligible benefits, since almost all of them are already compressed (*e.g.,* MP3 and VP9). Although it is possible to reduce network traffic by transcoding, scaling, or reducing the quality of multimedia content [42], we do not explore these potential optimizations in this work.

3.2.3 Content Validation

Delving deeper into the content downloaded by our volunteers, we discover a surprisingly high portion of useless content, particularly *broken* images. We define an image to be broken if it cannot be decoded by any of the three widely used image decoders: imghdr [12], Bitmap [23], and dwebp [7]. As shown in Table 4, 10.6% of images in TGdataset are broken, wasting 3.2% of all image traffic in our dataset (their average size is much smaller than that of correct images). Note that we also observe a small fraction of *blank* and *incomplete* images that we can decode, as well as a few *inconsistent* images that are actually not images, but we do not consider to obey our strict definition of correctness.

3.2.4 Traffic Filtering

As we note in § 2, existing mobile traffic proxies have adopted multiple strategies for traffic filtering. In this section, we investigate the potential of four particular filtering strategies by analyzing TGdataset.

Overnight traffic. Prior studies have observed that many smartphones generate data traffic late at night, even when users are not using the devices [52, 39, 34]. If we conservatively assume that our volunteers are asleep between 0–6 AM, then 11.4% of traffic in our dataset can potentially be filtered without noticeable impact on users. Based on this finding, we implemented a feature in TrafficGuard that allows users to specify a night time period during which cellular traffic is restricted (see § 5.3).

Background traffic. Users expect foreground apps to consume data since they are interactive, but background apps may also consume network resources. Although this is expected in some cases (*e.g.,* a user may stream music while also browsing the web), undesirable data consumption by background apps has become such a common complaint that numerous articles exist to help mitigate this problem [58, 3, 2, 10]. In TGdataset, we observe that 26.7% of cellular traffic is caused by background apps. To this end, we implemented dual filters in TrafficGuard specifically designed to reduce the network traffic of background apps (see § 5.3).

Malicious traffic. A recent measurement study of Google Play reveals that more than 25% of Android apps are malicious, including spammy, re-branded, and cloned apps [59]. We compare all the HTTP requests in TGdataset against a proprietary blacklist containing 29M links maintained by major Internet companies (including Baidu, Google, Microsoft, Symantec, Tencent, *etc.*), and find that 0.85% of requests were issued for malicious content. We addressed this issue in TrafficGuard by filtering out HTTP requests for blacklisted URLs.

Advertisement traffic. In addition to malicious content, we also find that 4.15% of HTTP requests in TGdataset were for ads. We determined this by comparing all the requested HTTP URLs in our dataset against a proprietary list of 102M known advertising URLs (similar to the well-known EasyList [1]). Ad blocking is a morally complicated practice, and thus we give TrafficGuard users the choice of whether to opt-in to ad filtering. Users' configuration data reveal that the majority (67%) of users have chosen to block ads. On the other hand, we did get pushback from a small number of advertisers; when this happened, usually we would remove the advertisers from our ad block list after verification.

3.2.5 Caching Strategies

Finally, we explore the feasibility of two common caching strategies. Unfortunately, we find neither technique offers satisfactory performance, which motives us to implement a more sophisticated caching strategy.

Name-based. Traditional web proxies like Squid [61] implement *name-based* caching of objects (*i.e.,* objects are indexed by their URLs). However, this approach is known to miss many opportunities for caching [38, 57, 51]. To make matters worse, we observe that over half of the content in TGdataset is not cacheable by Squid due to HTTP protocol issues. This situation is further exacerbated by the fact that many start-of-the-art HTTP libraries do not support caching at all [50]. Thus, although TrafficGuard uses Squid in the back-end cloud, we decided to augment it with an additional, object-level caching strategy (known as VBWC, see § 5.4).

HTTP ETag. The HTTP ETag [11] was introduced in HTTP/1.1 to mitigate the shortcomings of named-based caching. Unfortunately, the effectiveness of ETag is still limited by two constraints. First, as ETags are assigned arbitrarily by web servers, they do not allow clients to detect identical content served by multiple providers. This phenomenon is called content *aliasing* [55]. We observe that 14.16% of HTTP requests in TGdataset are for aliased content, corresponding to 7.28% of HTTP traffic. Second, we find that ETags are sparsely supported: only 5.76% of HTTP responses include ETags.

4 System Overview

Our measurement findings in § 3.2 provide useful guidelines for optimizing cellular traffic across apps. Additionally, we observe that some techniques used by prior systems (*e.g.,* text compression) are not useful in practice. These findings guide the design of TrafficGuard for optimizing users' cellular traffic.

This section presents an overview of TrafficGuard, which consists of a front-end mobile app on users' devices and a set of back-end services. Below, we present the basic components of each end, with an emphasis on how these components support various traffic optimization mechanisms. Additional details about specific traffic optimization mechanisms are explained in § 5.

4.1 Mobile App: The Client-side Support

The TrafficGuard mobile app is comprised of a user interface and a *child proxy*. The user interface is responsible for displaying cellular usage statistics, and allows users to configure TrafficGuard settings. The settings include enabling/disabling specific traffic optimization mechanisms, as well as options for specific mechanisms (the details are discussed in § 5). We also leverage the user interface to collect feedback from users, which help us continually improve the design of TrafficGuard.

The child proxy does the real work of traffic optimization on the client side. It intercepts incoming and outgoing HTTP requests at the cellular interface, performs computations on them, and forwards (some) requests to the back-end cloud via a customized VPN tunnel. As shown in Figure 3, the client-side VPN tunnel is implemented using the TUN virtual network-level device [26] that intercepts traffic from or injects traffic to the TCP/IP stack. HTTP GET requests [3] are captured by the child proxy, encapsulated, and then sent to the back-end cloud for further processing. Accordingly, the child proxy is responsible for receiving responses from the back-end.

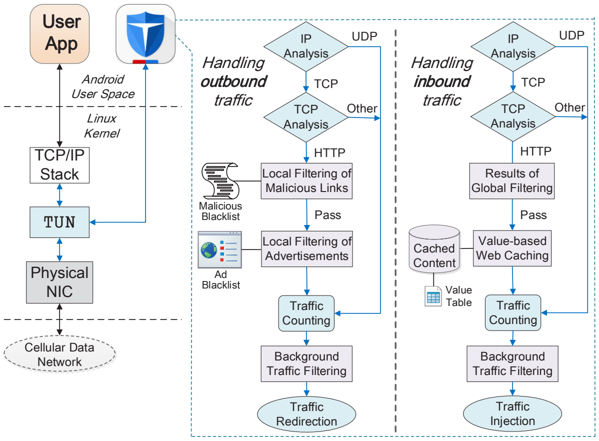


Figure 3: Basic design of the child proxy.

The mobile app provides client-side support for traffic optimization. First, it allows users to monitor and restrict cellular traffic at night and from background apps in a real-time manner. Users are given options to control how aggressively TrafficGuard filters these types of traffic. Second, it provides local filtering of malicious links and unwanted ads using two small blacklists of the most frequently visited malicious and advertising URLs. Requests for malicious URLs are dropped; users are given a choice of whether to enable ad blocking, in which case requests for ad-related URLs are also dropped.

Third, the child proxy acts as the client-side of a value-based web cache [55] (VBWC, see § 5.4 for details). At a high level, the child proxy maintains a key-value store that maps MD5 hashes to pieces of content. The back-end cloud may return "VBWC Hit" responses to the client that contain the MD5 hash of some content, rather than the content itself. In this case, the child proxy retrieves the content from the key-value store using the MD5 hash, and then locally constructs an HTTP response containing the cached content. The reconstructed HTTP response is then returned to the corresponding user app. This process is fully transparent to user apps.

4.2 Web Proxy: The Back-end Support

As shown in Figure 4, the cloud side of TrafficGuard consists of two components: a cluster of parent proxy servers that decapsulate users' HTTP GET requests and fetch content from the Internet; and a series of software middleboxes that process HTTP responses.

[3] Non-GET HTTP requests (*e.g.,* POST, HEAD, and PUT) and non-HTTP requests do not benefit from TrafficGuard's filtering and caching mechanisms, so the child proxy forwards them to the TCP/IP stack for regular processing. Furthermore, TrafficGuard makes no attempt to analyze SSL/TLS traffic for privacy reasons.

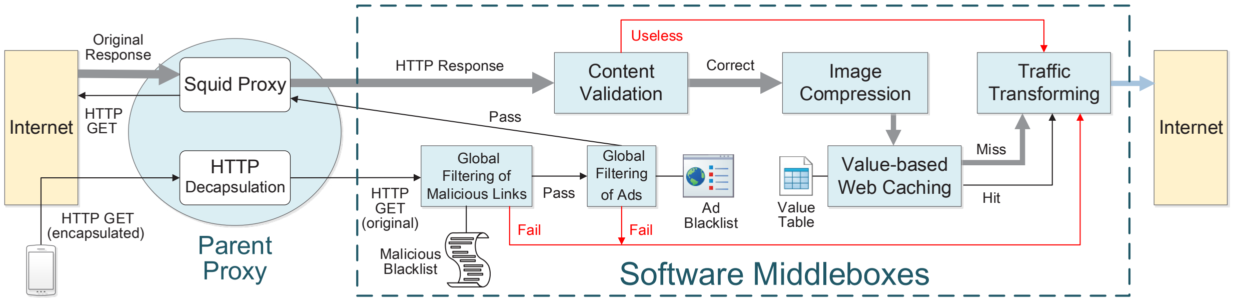


Figure 4: Cloud-side overview of TrafficGuard. HTTP requests are generally processed from left to right by a cluster of parent proxy servers and a series of software middleboxes implemented on top of Nginx.

Once an HTTP GET request sent by the child proxy is received, the parent proxy decapsulates it and extracts the *original* HTTP GET request. Next, middleboxes compare the original HTTP GET request against large blacklists of known malicious and ads-related URLs. Note that this HTTP GET request has passed the client-side filtering with small blacklists. Together, this two-level filtering scheme prevents TrafficGuard users from wasting memory loading large blacklists on their own devices. If a URL hits either blacklist, it is reported back to the mobile app so the user can be notified.

An HTTP request that passes the blacklist filters is forwarded to a Squid proxy, which fetches the requested content from the original source. The Squid proxy implements name-based caching of objects using an LRU (Least Recently Used) scheme, which helps reduce latency for popular objects. Once the content has been retrieved by Squid, it is further processed by middleboxes that validate content (§ 5.2) and compress images (§ 5.1).

Lastly, before the content is returned to users, it is indexed by VBWC (§ 5.4). VBWC maintains a separate index of content for every active user, which contains the MD5 hash of each piece of content recently downloaded by that user. For a given user, if VBWC discovers that some content is already indexed, it returns that MD5 in a "VBWC Hit" response to the mobile app, instead of the actual content. As described above, the child proxy then constructs a valid HTTP response message containing the cached content. Otherwise, the MD5 is inserted into the table and the actual content is sent to the user.

5 Mechanisms

This section presents the details of specific traffic optimization mechanisms in TrafficGuard. Since many of the mechanisms include user-configurable parameters, we gathered users' configuration data between Jul. 4–Dec. 27, 2014. This dataset is referred to as TGconfig.

5.1 Image Compression

Overview. Image compression is the most important traffic-reduction mechanism implemented by TrafficGuard, since our TGdataset shows that cellular traffic is dominated by images. Based on our observation that the majority of JPEGs have quality factors (QFs) that are excessively high for display on smartphone screens, TrafficGuard adaptively compresses JPEGs by reducing their QFs to an acceptable level. Additionally, TrafficGuard transcodes PNGs and GIFs to JPEGs with an acceptable QF. Note that TrafficGuard does *not* transcode PNGs with transparency data or animated GIFs, to avoid image distortion. TrafficGuard ignores WebP images, since they are already highly compressed.

TrafficGuard's approach to image compression has three advantages over alternative strategies. First, as JPEG is the dominant image format supported by almost all (>99% to our knowledge) user apps, TrafficGuard does not need to transcode images back to their original formats on the client side. Second, our approach costs only 10%–12% as much CPU as Flywheel's WebP-based transcoding method (see § 6.3). Finally, our approach does not alter the pixel dimensions of images. This is important because many UI layout algorithms (*e.g.*, CSS) are sensitive to the pixel dimensions of images, so rescaling images may break webpage and app UIs.

Categorizing Images. The challenge of implementing our adaptive QF reduction strategy is deciding how much to reduce the QFs of images. Intuitively, the QFs of large images can be reduced more than small images, since the resulting visual artifacts will be less apparent in larger images. Thus, following the approach of Ziproxy [28] (an open-source HTTP proxy widely used to compress images), we classify images into four categories according to their width (w) and height (h) in pixels:

- *Tiny* images contain < 5000 pixels, *i.e.*, $w \times h < 5000$.
- *Small* images include images with less than 50000 pixels (*i.e.*, $5000 \leq w \times h < 50000$), as well as "slim" images with less than 150 width or height pixels (*i.e.*, $w \times h \geq 5000 \wedge (w < 150 \vee h < 150)$).
- *Mid-size* images contain less than 250000 pixels, that is $50000 \leq w \times h < 250000 \wedge w \geq 150 \wedge h \geq 150$.
- *Large* images contain no less than 250000 pixels, that is $w \times h \geq 250000 \wedge w \geq 150 \wedge h \geq 150$.

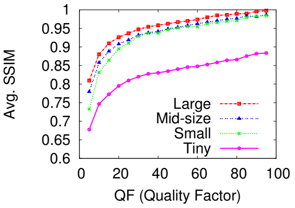


Figure 5: Average SSIM corresponding to consecutive QFs.

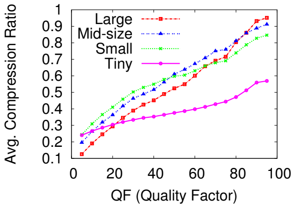


Figure 6: Average compression ratio corresponding to consecutive QFs.

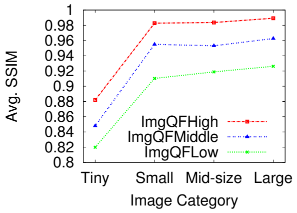


Figure 7: Average SSIM corresponding to the three QF schemes.

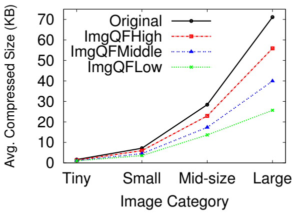


Figure 8: Average compressed size corresponding to the three QF schemes.

QF Reduction Scheme. After images are divided into the above four categories, we need to determine a proper *QF (reduction) scheme* for transcoding images in each category. Our goal is to maximize compression by reducing QF, while also minimizing the reductions of user-perceived image quality. To measure quality, we use Structural Similarity (SSIM) [60], which assesses the visual similarity between a compressed image and the original (1 means the two images are identical). Quantitatively, we calculate the SSIM and *compression ratio* ($= \frac{\text{Size of images after compression}}{\text{Size of images before compression}}$) corresponding to consecutive QFs, based on all the correct images in TGdataset. The results are plotted in Figure 5 and Figure 6.

Specifically, we define a QF scheme *ImgQFScheme* $= \{T, S, M, L\}$ to mean that tiny, small, mid-size, and large images are compressed to QF $= T$, S, M, and L, respectively. In practice, we constructed three QF schemes that vary from high compression, less quality to low compression, high quality: *ImgQFLow* $= \{30, 25, 25, 20\}$, *ImgQFMiddle* $= \{60, 55, 50, 45\}$, and *ImgQFHigh* $= \{90, 90, 85, 80\}$. We then compressed all the correct images in TGdataset using each scheme to evaluate their impact on image quality and size.

Figure 7 examines the impact of each QF scheme on image quality. Prior work has shown that image compression with SSIM ≥ 0.85 is generally considered acceptable by users [29]. As shown in Figure 7, all three QF schemes manage to stay above the 0.85 quality threshold for small, mid-size, and large images. The two cases where image quality becomes questionable concern tiny images, which are the hardest case for any compression strategy. Overall, these results suggest that in most cases, even the aggressive *ImgQFLow* scheme will produce images with an acceptable level of fidelity.

Figure 8 examines the image size reduction enabled by each QF scheme, as compared to the original images. As expected, more aggressive QF schemes provide more size reduction, especially for large images.

User Behavior. The mobile app of TrafficGuard allows users to choose their desired QF scheme. Users must select a scheme after they install TrafficGuard. The data in TGconfig reveal that 95.4% of users selected the *ImgQFMiddle* scheme. Also, qualitative feedback from TrafficGuard users suggests that they are satisfied with the quality of images while using the system.

5.2 Content Validation

As mentioned in § 3.2.3, TrafficGuard users encounter a non-trivial amount of broken images when using apps. The back-end cloud of TrafficGuard naturally notices most broken images during the image analysis, transcoding, and compression process. In these cases, the cloud simply discards the broken image and sends a "Broken Warning" response to the client. From the requesting app's perspective, broken images appear to be missing due to a network error, and are handled as such.

5.3 Traffic Filtering

In this section, we present the implementation details of the four types of filters employed by TrafficGuard. Most traffic filtering in our system occurs on the client side (in the child proxy), including first-level filtering of malicious URLs and ads, and throttling of overnight and background traffic. Only second-level filtering of malicious URLs and ads occurs on the cloud side.

Restricting overnight traffic. The mobile app of TrafficGuard automatically turns the user's cellular data connection off between the hours of t_1 and t_2, which are configurable by the user. This feature is designed to halt device traffic during the night, when the user is likely to be asleep. TrafficGuard pops-up a notification just before t_1, alerting the user that her cellular connection will be turned off in ten seconds. Unless the user explicitly cancels the action, her cellular data connection will not be resumed until t_2. According to TGconfig, nearly 20% of users have enabled the overnight traffic filter, and 84% of them adopt the default night duration of 0–6 AM.

Throttling background traffic. To prevent malicious or buggy apps from draining users' limited data plans, TrafficGuard throttles traffic from background apps. Specifically, the TrafficGuard app has a configurable *warning bound* (B_1) and a *disconnection bound* (B_2), with $B_2 \gg B_1$. TrafficGuard also maintains a count c of the total bytes transferred by background apps. If c increases to B_1, TrafficGuard notifies the user that background apps are consuming a significant volume of traffic. If c reaches B_2, another notification is created to alert the user that her cellular data connection will be closed in ten seconds. Unless the user explicitly cancels this action or manually re-opens the cellular data connection, her cellular data connection will not be resumed. After the user responds to the B_2 notification, c is reset to zero.

According to TGconfig, 97.6% of users have enabled the background traffic filter, indicating that users actually care about background traffic usage. Initially, we set the default warning bound $B_1 = 1.0\,\text{MB}$. However, we observed over 57% of users decreased B_1 to 0.5 MB, indicating that they wanted to be reminded of background traffic usage more frequently. Conversely, the initial disconnection bound was $B_2 = 5\,\text{MB}$, but 69% of users raised B_2 to 20 MB, implying that the initial default setting was too aggressive. Based on this implicit feedback, we changed the default values of B_1 and B_2 to 0.5 MB and 20 MB. In comparison, Microsoft Data Sense only maintains a disconnection bound (B_2) to restrict background traffic, and there is no default value provided.

Two-level filtering of malicious links and ads. To avoid wasting cellular traffic on unwanted content, TrafficGuard always prevents users from accessing malicious links, while giving users the choice of whether to opt-in to ad blocking. In § 4.1 and § 4.2, we have presented high-level design of the two-level filtering. Here we talk about two more nuanced implementation issues.

The first issue is about the sizes of the local, small blacklists. Both lists have to be loaded in memory by the child proxy for quick searching, so they must be much shorter than the cloud-side large blacklists (which contain 29M malicious URLs and 102M ads-related URLs). To balance memory overhead with effective local traffic filtering, we limit the maximum size of the local blacklists to 40 MB. Consequently, the local blacklists usually contain around 1M links in total, which we observe are able to identify 72%–78% of malicious and ads links.

The second issue concerns updates to blacklists. As mentioned in § 3.2.4, the large blacklists are maintained by an industrial union that typically updates them once per month. Accordingly, the TrafficGuard cloud automatically creates updated small blacklists and pushes them to mobile users.

5.4 Value-based Web Caching (VBWC)

Early in 2003, Rhea *et al.* proposed VBWC to overcome the shortcomings of traditional HTTP caching [55]. The key idea of VBWC is to index objects by their *hash values* rather than their URLs, since an object may have many *aliases*. VBWC has a much better hit rate than HTTP caching because it handles aliased content. However, prior to TrafficGuard, VBWC has not been widely deployed in practice due to two problems: 1) the *complexity* of segmenting an object into KB-sized blocks and choosing proper block boundaries; 2) its *incompatibility* with the HTTP protocol, since VBWC requires that the proxy and the client maintain significant state information, *i.e.,* a mapping from hash values to cached content.

Reducing complexity. To determine whether TrafficGuard's VBWC implementation should segment content into blocks (and if so, at what granularity), we conduct trace-driven simulations using the content in TGdataset. Specifically, we played back each user's log of requests, and inserted the content into VBWC using 8 KB, 32 KB, 128 KB, and full content segmentation strategies. To determine segment boundaries, we ran experiments with simple fixed-size segments [55] and variable-sized, Rabin-fingerprinting based segments [53]. We also examined the handprinting-based approach that combines Rabin-fingerprinting and deterministic sampling [48].

Through these simulations, we discovered that 13% of HTTP requests would hit the VBWC cache if we stored content whole, *i.e.,* with no segmentation. Surprisingly, even if we segmented content into 8 KB blocks using the Rabin-fingerprinting (the most aggressive caching strategy we evaluated), the hit rate only increased to 15%. The handprinting-based approach exhibited similar performance to Rabin-fingerprinting when a typical number ($k = 4$) of handprint samples are selected, while incurring a bit lower computation overhead. By carefully analyzing the cache-hit results, we find that the whole-content hashing is good enough for two reasons: 1) images dominate the size of cache-hit objects in TGdataset; 2) there are almost no partial matches among images. Thus, we conclude that a simple implementation of content-level VBWC is sufficient to achieve high hit rates.

Addressing incompatibility. As discussed above, VBWC is incompatible with standard HTTP clients and proxies. Fortunately, we have complete control over the TrafficGuard system, particulary the cloud-client paired proxies, which enabled us to implement VBWC. The front-end child proxy takes care of encapsulating HTTP requests from user apps and decapsulating responses from the back-end cloud, meaning that VBWC is transparent to user apps. In practice, the mobile app of TrafficGuard maintains a 50-MB content cache on the client's file system, along with an in-memory table mapping content hashes to filenames that is a few KB large.

Ideally, every change to the cloud-side mapping table triggers a change to the client-side mapping table accordingly. But in practice, for various reasons (*e.g.,* network packet loss) this pair of tables may be different at some time, so we need to synchronize them with proper overhead. In TrafficGuard, the client-side mapping table is loosely synchronized with the cloud-side mapping table on an hourly basis, making the synchronization traffic negligible and VBWC mostly effective.

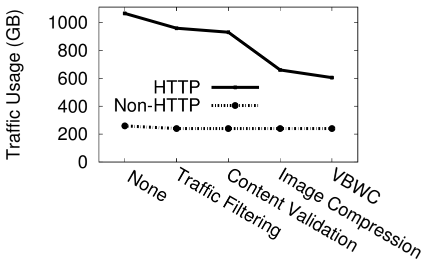


Figure 9: Total cellular traffic usage optimized by each mechanism.

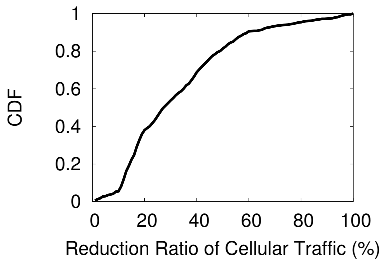


Figure 10: Distribution of users' cellular traffic reduction ratios.

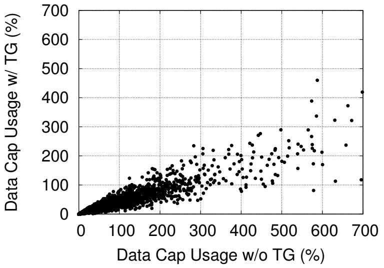


Figure 11: Users' cellular traffic usage relative to their data caps.

6 Evaluation

In this section, we evaluate the traffic reduction, system overhead, and latency penalty brought by TrafficGuard.

6.1 Data Collection and Methodology

We evaluate the performance of TrafficGuard using both real-system logs and trace-driven simulations. We collect working logs from TrafficGuard's back-end cloud servers between Dec. 21–27, 2014, which include traces of 350M HTTP requests issued from 0.6M users, as well as records of CPU and memory utilization over time on the cloud servers. We refer to this dataset as TGworklog.

On the other hand, as the client-side traffic optimization mechanisms mainly help users reduce traffic by suppressing unwanted requests, it is not possible to accurately record the corresponding saved traffic (which never occurred in reality). Instead, we rely on trace-driven simulations using TGdataset to estimate the client-side and overall traffic savings. In addition, we report real-world traffic reduction results using TGworklog in Appendix B, which mainly record the cloud-side traffic savings.

6.2 Traffic Reduction

Client-side. First, we examine the effectiveness of TrafficGuard's client-side mechanisms at reducing traffic. In TGdataset, 11.4% of cellular traffic is transferred at night, and according to TGconfig, 20% of users have enabled overnight traffic filtering. Thus, we estimate that users eliminate 2.3% (= 11.4% × 20%) of cellular traffic using the overnight traffic filter.

Moreover, we observe that 1% of users in TGdataset regularly exceed the disconnection bound $B_2 = 20$ MB per day of background traffic. The resulting overage traffic amounts to 5.33% of cellular traffic. In TGconfig, 97.6% of users have enabled background traffic filtering. Therefore, we estimate that the background traffic filter reduces cellular traffic by 5.2% (= 5.33% × 97.6%). Note that this background traffic saving is an *underestimation*, since we do not take the potential effect of B_1 (= 0.5 MB, the warning bound) into account.

Additionally, in TGdataset malicious content accounts for 0.8% of HTTP traffic while ads account for 4%. According to TGconfig, 67% of users have chosen to drop ads. Consequently, after all malicious content and unwanted ads are filtered, 3.48% $(= 0.8\% + 4\% \times 67\%)$ of HTTP traffic can be saved. This is equal to 2.8% $(= 3.48\% \times 80.4\%)$ of total cellular traffic.

Overall. Next, we evaluate how much traffic TrafficGuard is able to reduce overall through trace-driven simulations. Specifically, we play back all the requests in TGdataset, and record how many bytes are saved by each mechanism: traffic filtering, content validation, image compression, and VBWC. As shown in Figure 9, TrafficGuard is able to reduce HTTP traffic by 43% and non-HTTP traffic by 7.4% when all four mechanisms are combined. In summary, the overall cellular traffic usage is reduced by 36%, from 1324 GB to 845 GB.

As expected, image compression is the most important mechanism when used in isolation. 38% of the image traffic is reduced by our implemented adaptive quality reduction approach. In other words, our approach saves a comparable portion $(27\% = 38\% \times 71\%)$ of HTTP traffic as compared to Flywheel's WebP-based transcoding method, at a small fraction of the CPU cost (see § 6.3).

To understand how traffic savings are spread across users, we plot the distribution of cellular traffic reduction ratios for our users in Figure 10. We observe that 55% of users saved over a quarter of cellular traffic, and 20% users saved over a half (most of whom benefit a lot from traffic filtering and VBWC). These results demonstrate that most users received significant traffic savings.

Using TrafficGuard's built-in user-feedback facility, we asked users to report their cellular data caps. 95% of the long-term volunteers in TGdataset reported their caps to us. Using this information, we plot Figure 11, which shows the percentage of each user's data cap that would be used with and without TrafficGuard (again, based on trace-driven simulations). We observe that 58.2% of users exceed their usage caps under normal circumstances, and that TrafficGuard grants significant practical benefits for these users, *e.g.,* users who would normally be using 200%–300% of their allocation (and thus pay overage fees) are able to stay below 100% usage with TrafficGuard. Overall, TrafficGuard reduces the number of users who exceed their data caps by 10.7 times.

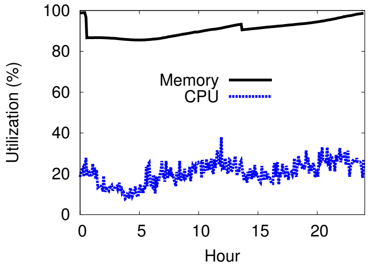


Figure 12: CPU and memory overhead of the back-end servers.

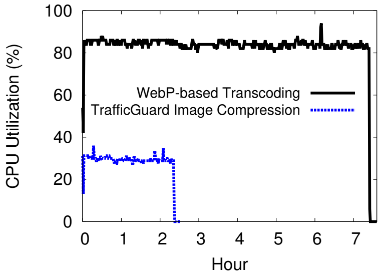


Figure 13: CPU overhead of different image compression strategies.

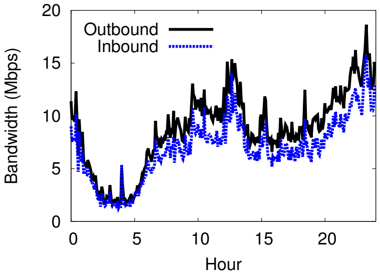


Figure 14: Inbound and outbound bandwidth for back-end servers.

Table 5: Top-10 applications served by TrafficGuard, ordered by popularity and by greatest traffic reduction.

By User Ratio (UR)			By Traffic Saving Ratio (TSR)		
App Name	**UR**	**TSR**	**App Name**	**UR**	**TSR**
WeChat	74%	22%	Android Browser	0.11%	84%
QQ	66%	22%	Zhihu Q&A	0.15%	81%
Baidu Search	29%	21%	iAround	0.03%	63%
Taobao	23%	42%	No.1 Store	0.26%	61%
QQBrowser	22%	27%	Baidu News	0.45%	57%
Sogou Pinyin	20%	12%	Tiexue Military	0.01%	56%
Baidu Browser	16%	30%	WoChaCha	0.34%	54%
Toutiao News	14%	22%	Mogujie Store	0.91%	53%
Sohu News	10%	30%	Koudai Store	0.26%	53%
QQ Zone	10%	33%	Papa Photo	0.02%	52%

At last, we wonder how TrafficGuard's traffic reduction gains are spread across user apps. Table 5 lists the top-10 apps ordered by popularity (the fraction of users with the app) as well as by the fraction of traffic eliminated. We observe that TrafficGuard is able to eliminate 12%–42% of traffic for popular apps, but that the apps with the greatest traffic savings (52%–84%) tend to be unpopular. This indicates that the developers of popular apps may already be taking steps to optimize their network traffic, while most unpopular apps can hardly become mobile-friendly in the near future.

6.3 System Overhead

Cloud-side overhead. The major cost of operating TrafficGuard lies in provisioning back-end cloud servers and supplying them with bandwidth. TrafficGuard has been able to support ∼0.2M users who send ∼90M requests per day using only 23 commodity servers (HP ProLiant DL380). The configuration of each server is: 2*4-core Xeon CPU E5-2609 @2.50GHz, 4*8-GB memory, and 6*300-GB 10K-RPM SAS disk (RAID-6).

Figure 12 illustrates the CPU/memory utilization of cloud servers on a typical day. Mainly thanks to our lightweight image compression strategy, the CPU utilization stays below 40%. Further, to compare the computation overhead of our image compression strategy with Flywheel's WebP-based transcoding (based on the cwebp [5] encoder), we conduct offline experiments on two identical server machines using 1M correct images randomly picked from TGdataset as the workload. Images are compressed one by one without intermission. The results in Figure 13 confirm that the computation overhead (= average CPU utilization × total running time) of TrafficGuard image compression is only a small portion (10%–12%) of that of WebP-based transcoding.

Memory utilization is typically >90% since content is in-memory cached whenever possible. Using a higher memory capacity, say 1 TB per server, can accelerate the back-end processing and thus decrease the corresponding latency penalty. Nonetheless, as shown in Figure 16, the back-end processing latency constitutes only a minor portion of the total latency penalty, so we do not consider extending the memory capacity in the short term.

Figure 14 reveals the inbound/outbound bandwidth for back-end servers. Interestingly, we observe that a back-end server uses more outbound bandwidth than inbound, though inbound traffic has been optimized. This happens because the back-end has a 38% cache hit rate (with 4 TB of disk cache), so many objects are downloaded from the Internet once but then downloaded by many clients.

Client-side overhead. The client-side overhead of TrafficGuard comes from three sources: memory, computation, and battery usage. The memory usage is modest, requiring 40 MB for local blacklists, and 10–20 MB for the VBWC mapping table. Similarly, while running on a typical (8-core ARM CPU @1.7GHz) Android smartphone, TrafficGuard's single-core CPU usage is generally below 20% when the cellular modem is active, and almost zero when the network is inactive.

To understand the impact of TrafficGuard on battery life, we record the battery power consumption of its mobile app when the child proxy is processing data packets. As shown in Figure 15, its working-state battery power is 93 mW on average, given that the battery capacity of today's smartphones lies between 5–20 Wh and their working-state battery power lies between 500 mW and a few watts [35, 45]. We also conduct micro-benchmarks to examine specific facets of TrafficGuard's battery consumption (see Appendix C for details). Micro-benchmark results illustrate that TrafficGuard can effectively reduce the battery consumption of user apps by optimizing their traffic.

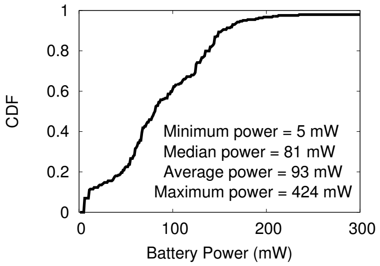


Figure 15: Distribution of client-side battery power consumption.

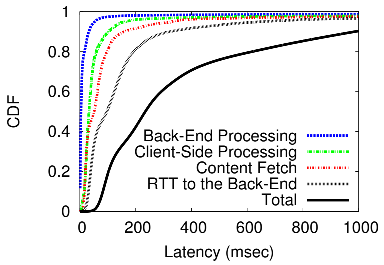


Figure 16: Latency for each phase content processing and retrieval.

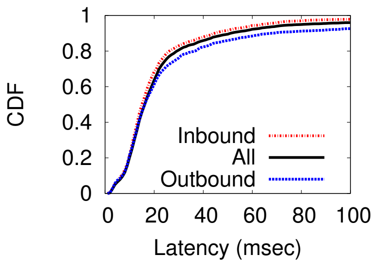


Figure 17: Latency for clients' processing data packets.

6.4 Latency Penalty

As TrafficGuard forwards HTTP GET requests to a back-end proxy rather than directly to the source, it may add response latency to clients' requests. In addition, client-side packet processing by the child proxy also brings extra latency. To put the latency penalty into perspective, first, we note three mitigating factors that effectively reduce latency: 1) TrafficGuard filters out ∼10.3% of requests locally, which eliminates all latency except for client-side processing; 2) the ∼21.2% of traffic that is not owing to HTTP GET requests is delivered over the Internet normally, thus only incurring the latency penalty for client-side processing; and 3) 38% of HTTP GET requests hit the back-end Squid cache, thus eliminating the time needed to fetch the content from the Internet.

Next, to understand TrafficGuard's latency penalty in the worst-case scenario (unfiltered HTTP GETs that do not hit the Squid cache), we examine latency data from TGworklog. Figure 16 plots the total latency of requests that go through the TrafficGuard back-end and miss the cache, as well as the individual latency costs of four aspects of the system: 1) processing time on the client side, 2) processing time in the back-end, 3) time for the back-end to fetch the desired content, and 4) the RTT from the client to the back-end. Figure 16 shows that both client-side processing and back-end processing add little delay to requests. Instead, the majority of delay comes from fetching content, and the RTT from clients to the back-end cloud. Interestingly, Figure 17 reveals that the average processing time of an outbound packet is longer than that of an inbound packet, although outbound packets are usually smaller than inbound packets. This is because the client-side filtering of malicious links and ads is the major source of client-side latency penalty.

In the worst-case scenario, we see that TrafficGuard does add significant latency to user requests. If we conservatively assume that clients can fetch content with the same latency distribution as Baidu's servers, then TrafficGuard adds 131 ms of latency in the median case and 474 ms of latency in the average case. However, if we take into account the three mitigating factors listed at the beginning of this section (which all reduce latency), the median latency penalty across all traffic is reduced to merely 53 ms, and the average is reduced to 282 ms.

7 Conclusion

Traffic optimization is a common desire of today's cellular users, carriers, and service developers. Although several existing systems can optimize the cellular traffic for specific apps (typically web browsers), cross-app systems are much rarer, and have not been comprehensively studied. In this paper, we share our design approach and implementation experiences in building and maintaining TrafficGuard, a real-world cross-app cellular traffic optimization system used by 10 million users.

To design TrafficGuard, we took a measurement-driven methodology to select optimization strategies that are not only high-impact (*i.e.,* they significantly reduce traffic) but also efficient, easy to implement, and compatible with heterogenous apps. This methodology led to some surprising findings, including the relative ineffectiveness of text compression. Real-world performance together with trace-driven experiments indicate that our system meets its stated goal of reducing traffic (by 36% on average), while also being efficient (23 commodity servers are able to handle the entire workload). In the future, we plan to approach cellular carriers about integrating TrafficGuard into their networks, since this will substantially decrease latency penalties for users and simplify the overall design of the system.

Acknowledgments. We wish to thank the following people for their contributions to the system or the paper. Changqing Han, Junyi Shao, Xuefeng Luo, Min Guo, Cheng Peng, Tianwei Wen, and Zhefu Jiang helped develop the system. Yuxuan Yan and Jian Chen helped evaluate the client-side latency penalty and battery consumption. Our shepherd Matt Welsh guided the preparation of the camera-ready paper.

This work is supported by the High-Tech R&D ("863 – China Cloud") Program of China under grant 2015AA01A201, China NSF under grants 61432002 and 61471217, NSF under grants IIS-1321083, ECCS-1247944, CNS-1526638 and CNS-1319019, and CCF-Tencent Open Fund under grant IAGR20150101.

References

[1] Adblock Plus EasyList for ad blocking. http://easylist.adblockplus.org.

[2] Android OS background data increase since 4.4.4 update. http://forums.androidcentral.com/moto-g-2013/422075-android-os-background-data-increase-since-4-4-4-update-please-help.html.

[3] Android OS is continuously downloading something in the background. http://android.stackexchange.com/questions/28100/android-os-is-continuously-downloading-something-in-the-background-how-can-i.

[4] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 2014-2019 White Paper. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html.

[5] cwebp – Compress an image file to a WebP file. http://developers.google.com/speed/webp/docs/cwebp.

[6] Data Sense for Windows Phone apps. http://www.windowsphone.com/en-us/how-to/wp8/connectivity/use-data-sense-to-manage-data-usage.

[7] dwebp – Decompress a WebP file to an image file. http://developers.google.com/speed/webp/docs/dwebp.

[8] Google Safe Browsing. http://developers.google.com/safe-browsing.

[9] Google to websites: Be mobile-friendly or get buried in search results. http://mashable.com/2015/04/21/google-mobile-search-2/#UbuurRKFaPqU.

[10] How to Minimize Your Android Data Usage and Avoid Overage Charges. http://www.howtogeek.com/140261/how-to-minimize-your-android-data-usage-and-avoid-overage-charges.

[11] HTTP ETag. http://en.wikipedia.org/wiki/HTTP_ETag.

[12] imghdr – Determine the type of an image. http://docs.python.org/2/library/imghdr.html.

[13] Institutional review board (IRB). https://en.wikipedia.org/wiki/Institutional_review_board.

[14] Netequalizer Bandwidth Shaper. http://www.netequalizer.com.

[15] Onavo Extend for Android. http://www.onavo.com/apps/android_extend.

[16] Opera Max. http://www.operasoftware.com/products/opera-max.

[17] Opera Max, China's version. http://www.oupeng.com/max.

[18] Opera Turbo mobile web proxy. http://www.opera.com/turbo.

[19] Packeteer WAN Optimization Solutions. http://www.bluecoat.com/packeteer.

[20] Peribit WAN Optimization. http://www.juniper.net.

[21] QQBrowser. http://browser.qq.com.

[22] Riverbed Networks. http://www.riverbed.com.

[23] System.Drawing.Bitmap class in the .NET Framework. http://msdn.microsoft.com/library/system.drawing.bitmap(v=vs.110).aspx.

[24] The State Of Digital Experience Delivery, 2015. https://www.forrester.com/The+State+Of+Digital+Experience+Delivery+2015/fulltext/-/E-RES120070.

[25] UCBrowser. http://www.ucweb.com.

[26] Universal TUN/TAP device driver. http://www.kernel.org/doc/Documentation/networking/tuntap.txt.

[27] WebP: A new image format for the Web. http://developers.google.com/speed/webp.

[28] Ziproxy: the HTTP traffic compressor. http://ziproxy.sourceforge.net.

[29] AGABABOV, V., BUETTNER, M., CHUDNOVSKY, V., COGAN, M., GREENSTEIN, B., MCDANIEL, S., PIATEK, M., SCOTT, C., WELSH, M., AND YIN, B. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *Proc. of NSDI* (2015), USENIX, pp. 367–380.

[30] AGARWAL, B., AKELLA, A., ANAND, A., BALACHANDRAN, A., CHITNIS, P., MUTHUKRISHNAN, C., RAMJEE, R., AND VARGHESE, G. EndRE: An End-System Redundancy Elimination Service for Enterprises. In *Proc. of NSDI* (2010), USENIX, pp. 419–432.

[31] ANAND, A., GUPTA, A., AKELLA, A., SESHAN, S., AND SHENKER, S. Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination. In *Proc. of SIGCOMM* (2008), ACM, pp. 219–230.

[32] ANAND, A., MUTHUKRISHNAN, C., AKELLA, A., AND RAMJEE, R. Redundancy in Network Traffic: Findings and Implications. In *Proc. of SIGMETRICS* (2009), ACM, pp. 37–48.

[33] ANAND, A., SEKAR, V., AND AKELLA, A. SmartRE: An Architecture for Coordinated Network-wide Redundancy Elimination. In *Proc. of SIGCOMM* (2009), ACM, pp. 87–98.

[34] AUCINAS, A., VALLINA-RODRIGUEZ, N., GRUNENBERGER, Y., ERRAMILLI, V., PAPAGIANNAKI, K., CROWCROFT, J., AND WETHERALL, D. Staying Online While Mobile: The Hidden Costs. In *Proc. of CoNEXT* (2013), ACM, pp. 315–320.

[35] CARROLL, A., AND HEISER, G. An Analysis of Power Consumption in a Smartphone. In *Proc. of USENIX ATC* (2010).

[36] CUI, Y., LAI, Z., WANG, X., DAI, N., AND MIAO, C. QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services. In *Proc. of MobiCom* (2015), ACM, pp. 592–603.

[37] FALAKI, H., LYMBEROPOULOS, D., MAHAJAN, R., KANDULA, S., AND ESTRIN, D. A First Look at Traffic on Smartphones. In *Proc. of IMC* (2010), ACM, pp. 281–287.

[38] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext transfer protocol – HTTP/1.1, 1999.

[39] HUANG, J., QIAN, F., MAO, Z., SEN, S., AND SPATSCHECK, O. Screen-Off Traffic Characterization and Optimization in 3G/4G Networks. In *Proc. of IMC* (2012), ACM, pp. 357–364.

[40] ISAACMAN, S., AND MARTONOSI, M. Low-Infrastructure Methods to Improve Internet Access for Mobile Users in Emerging Regions. In *Proc. of WWW* (2011), ACM, pp. 473–482.

[41] JOHNSON, D., PEJOVIC, V., BELDING, E., AND VAN STAM, G. Traffic Characterization and Internet Usage in Rural Africa. In *Proc. of WWW* (2011), ACM, pp. 493–502.

[42] Li, Z., Huang, Y., Liu, G., Wang, F., Zhang, Z.-L., and Dai, Y. Cloud Transcoder: Bridging the Format and Resolution Gap between Internet Videos and Mobile Devices. In *Proc. of NOSSDAV* (2012), ACM, pp. 33–38.

[43] Li, Z., Jin, C., Xu, T., Wilson, C., Liu, Y., Cheng, L., Liu, Y., Dai, Y., and Zhang, Z.-L. Towards Network-level Efficiency for Cloud Storage Services. In *Proc. of IMC* (2014), ACM, pp. 115–128.

[44] Li, Z., Wilson, C., Xu, T., Liu, Y., Lu, Z., and Wang, Y. Offline Downloading in China: A Comparative Study. In *Proc. of IMC* (2015), ACM, pp. 473–486.

[45] Liu, Y., Xiao, M., Zhang, M., Li, X., Dong, M., Ma, Z., Li, Z., and Chen, S. GoCAD: GPU-assisted Online Content Adaptive Display Power Saving for Mobile Devices in Internet Streaming. In *Proc. of WWW* (2016), ACM.

[46] Lumezanu, C., Guo, K., Spring, N., and Bhattacharjee, B. The Effect of Packet Loss on Redundancy Elimination in Cellular Wireless Networks. In *Proc. of IMC* (2010), ACM, pp. 294–300.

[47] Naylor, D., Schomp, K., Varvello, M., Leontiadis, I., Blackburn, J., Lopez, D., Papagiannaki, K., Rodriguez, P., and Steenkiste, P. multi-context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proc. of SIGCOMM* (2015), ACM, pp. 199–212.

[48] Pucha, H., Andersen, D., and Kaminsky, M. Exploiting Similarity for Multi-Source Downloads Using File Handprints. In *Prof. of NSDI* (2007), USENIX.

[49] Qian, F., Huang, J., Erman, J., Mao, Z., Sen, S., and Spatscheck, O. How to Reduce Smartphone Traffic Volume by 30%? In *Proc. of PAM* (2013), Springer, pp. 42–52.

[50] Qian, F., Quah, K., Huang, J., Erman, J., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O. Web Caching on Smartphones: Ideal vs. Reality. In *Proc. of MobiSys* (2012), ACM, pp. 127–140.

[51] Qian, F., Sen, S., and Spatscheck, O. Characterizing Resource Usage for Mobile Web Browsing. In *Proc. of MobiSys* (2014), ACM, pp. 218–231.

[52] Qian, F., Wang, Z., Gao, Y., Huang, J., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O. Periodic Transfers in Mobile Applications: Network-wide Origin, Impact, and Optimization. In *Proc. of WWW* (2012), ACM, pp. 51–60.

[53] Rabin, M. *Fingerprinting by Random Polynomials*. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[54] Rao, A., Kakhki, A., Razaghpanah, A., Tang, A., Wang, S., Sherry, J., Gill, P., Krishnamurthy, A., Legout, A., Mislove, A., and Choffnes, D. Using the Middle to Meddle with Mobile. *Tech. Report NEU-CCS-2013-12-10, CCIS, Northeastern University*.

[55] Rhea, S., Liang, K., and Brewer, E. Value-based Web Caching. In *Proc. of WWW* (2003), ACM, pp. 619–628.

[56] Sherry, J., Lan, C., Popa, R., and Ratnasamy, S. Blind-Box: Deep Packet Inspection over Encrypted Traffic. In *Proc. of SIGCOMM* (2015), ACM, pp. 213–226.

[57] Spring, N., and Wetherall, D. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proc. of SIGCOMM* (2000), ACM, pp. 87–95.

[58] Vergara, E., Sanjuan, J., and Nadjm-Tehrani, S. Kernel Level Energy-efficient 3G Background Traffic Shaper for Android Smartphones. In *Prof. of the 9th International Wireless Communications and Mobile Computing Conference (IWCMC)* (2013), IEEE, pp. 443–449.

[59] Viennot, N., Garcia, E., and Nieh, J. A Measurement Study of Google Play. In *Proc. of SIGMETRICS* (2014), ACM, pp. 221–233.

[60] Wang, Z., Bovik, A., Sheikh, H., and Simoncelli, E. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing 13*, 4 (2004), 600–612.

[61] Wessels, D. *Squid: The Definitive Guide*. O'Reilly Media, Inc., 2004.

[62] Woo, S., Jeong, E., Park, S., Lee, J., Ihm, S., and Park, K. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *Proc. of MobiSys* (2013), ACM, pp. 319–332.

[63] Zhai, E., Chen, R., Wolinsky, D. I., and Ford, B. An Untold Story of Redundant Clouds: Making Your Service Deployment Truly Reliable. In *Proc. of HotDep* (2013), ACM.

[64] Zhai, E., Chen, R., Wolinsky, D. I., and Ford, B. Heading Off Correlated Failures through Independence-as-a-Service. In *Proc. of OSDI* (2014), USENIX, pp. 317–334.

A Performance Analysis of Cross-App RE Techniques and VBWC

In this appendix, we discuss alternative caching strategies to VBWC, and motivate our ultimate selection of VBWC for TrafficGuard. A common approach to optimizing cross-app cellular traffic is called redundancy elimination (RE) that removes repeated data transfer [57, 31, 33, 32, 30, 46, 62, 49, 63, 64]. It can be deployed in ISP middleboxes [14, 19, 20, 22], on Internet routers [31], or in an end-to-end manner (*i.e.,* EndRE [30]).

RE relies on a pair of synchronized packet caches deployed at each end of a network path [57]. At one end, the sender (*e.g.,* the parent proxy in TrafficGuard) compresses data packets by replacing sequences of bytes that have appeared in previous packets with fixed-size pointers. At the other end, the receiver (*e.g.,* the child proxy in TrafficGuard) decodes data packets by following the pointers and replacing compressed data with the cached original data.

Informed Marking RE. Lumezanu *et al.* point out that TCP/IP packet loss (also including the cases of packet disorder and retransmission) can considerably degrade the performance of RE in cellular networks, and thus propose the enhanced *Informed Marking* RE algorithm [46]. To quantitatively understand the effect of Informed Marking RE, we conduct trace-driven simulations based on TGdataset. The simulation results indicate that merely 4.6% of HTTP traffic and 1.5% of non-HTTP traffic can be saved.

EndRE vs. VBWC. EndRE (*i.e.,* end-to-end RE) usually runs above the transport layer, so it is immune to TCP/IP packet loss. Following the design principle in [30], we simulate EndRE on TGdataset, and observe that as high as 10.2% of HTTP traffic can be saved — even better than the savings of VBWC (9%).

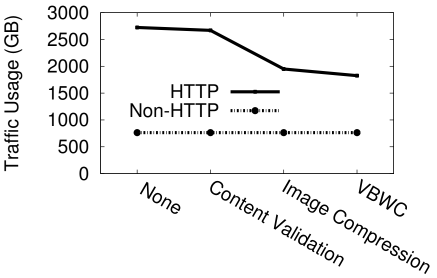


Figure 18: Real-world cellular traffic usage optimized by each mechanism.

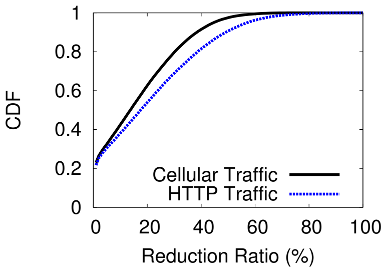


Figure 19: Distribution of real-world traffic reduction ratios across users.

Unfortunately, EndRE incurs much higher complexity in terms of implementation, computation, and cache maintenance. First, it is fairly straightforward to implement VBWC (refer to § 5.4), but implementing EndRE is not simple. Second, a poorly-provisioned EndRE client needs 60 MB of memory [30], which is even larger than the total client-side memory overhead (< 60 MB) of TrafficGuard. Even worse, the server-side memory overhead of EndRE can be hundreds of times higher than that of VBWC. Third, EndRE needs to maintain simultaneous TCP connections to guarantee cache consistency [30], while VBWC uses soft-state and is robust to temporary cache inconsistency.

Collaborative Caching. By conducting a week-long measurement of 3G traffic at a large cellular ISP in South Korea, Woo *et al.* observe that simple TCP-level RE can save 27%–42% of traffic with a *collaborative cache* of 512 GB [62]. However, such saving ratios can only be acquired at a centralized vantage point in the cellular backhaul networks, rather than an end point from a cellular user's perspective. In fact, the dataset collected by Woo *et al.* does not contain the identifiers (*e.g.,* IMEI or IMSI) of user devices, thus making the per-user analysis of traffic saving impossible.

B Real-world Traffic Reduction Results

As mentioned in § 6.1, the real-world working logs of TrafficGuard (*i.e.,* TGworklog) do not include the detailed information of filtered traffic (since they never occurred in reality). In other words, TGworklog mainly records the known traffic reduction results on the cloud side, through the traffic optimization mechanisms of content validation, image compression, and VBWC.

Meanwhile, as we note in § 4.1, the mobile app of TrafficGuard provides the user with an interface for displaying various cellular usage statistics, particularly the traffic saving statistics. Here the traffic saving statistics are also extracted from the real-world working logs of TrafficGuard, so they are less than the overall traffic savings studied in § 6.2.

As shown in Figure 18, HTTP traffic is reduced by 33%, while non-HTTP traffic cannot be reduced since non-HTTP traffic is not forwarded and processed by the back-end servers of TrafficGuard. In total, 26% of cellular traffic is reduced according to TGworklog.

In detail, we plot the distribution of real-world cellular/HTTP traffic reduction ratios across users in Figure 19. We observe that 38% of users saved over a quarter of HTTP traffic, and 10% of users saved over a half. In comparison, 29% of users saved over a quarter of cellular traffic, and merely 2.5% users saved over a half.

C Micro-Benchmark Results of TrafficGuard's Battery Consumption

To understand specific facets of TrafficGuard's battery consumption, we conduct micro-benchmarks on the client side with three popular, diverse user apps: the stock Android Browser, WeChat (the most popular app in China, similar to WhatsApp), and Youku (China's equivalent of YouTube). In each case, we drove the app for five minutes with and without TrafficGuard enabled while connected to a 4G network.

Figures 20, 21, and 22 show the battery usage in each experiment. Meanwhile, Figures 23, 24, and 25 depict the corresponding CPU usage; Figures 26, 27, and 28 plot the corresponding memory usage. All these results reveal that in cases where TrafficGuard can effectively reduce network traffic (*e.g.,* while browsing the web), it also saves battery life or has little impact on battery life, because the user app needs to process less traffic; accordingly, TrafficGuard does not increase CPU/memory usage on the whole. However, in cases where TrafficGuard can hardly reduce any traffic (*e.g.,* Youku video streaming), it reduces battery life and increases CPU/memory usage. Thus, we are planning to improve the design of TrafficGuard, in order that it can recognize and bypass the traffic from audio/video streams.

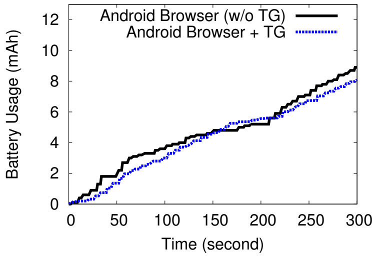


Figure 20: Battery usage of Android Browser with and without TrafficGuard (abbreviated as TG).

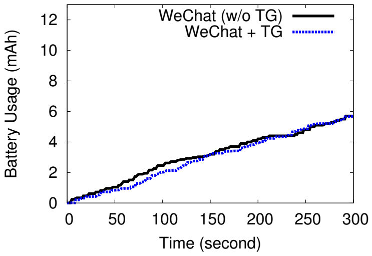


Figure 21: Battery usage of WeChat with and without TrafficGuard.

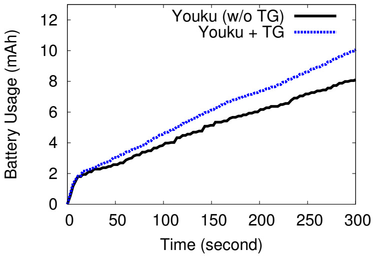


Figure 22: Battery usage of Youku with and without TrafficGuard.

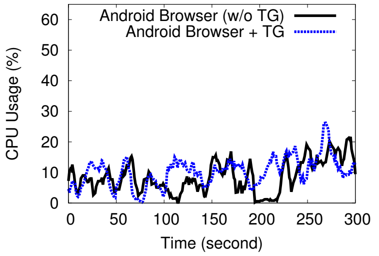


Figure 23: CPU usage of Android Browser w/ and w/o TrafficGuard.

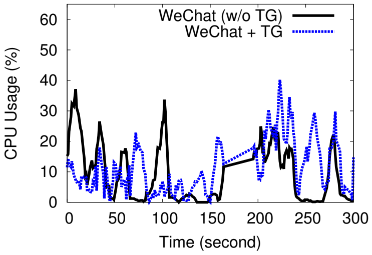


Figure 24: CPU usage of WeChat with and without TrafficGuard.

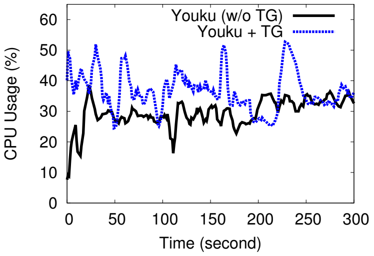


Figure 25: CPU usage of Youku with and without TrafficGuard.

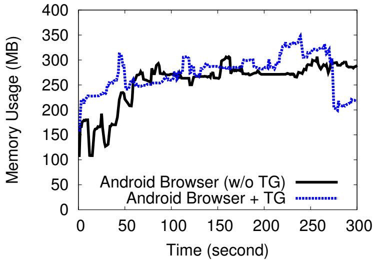


Figure 26: Memory usage of Android Browser with and without TrafficGuard.

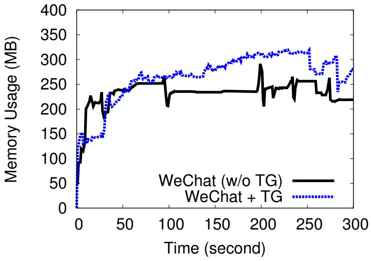


Figure 27: Memory usage of WeChat with and without TrafficGuard.

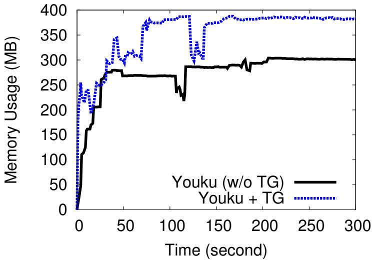


Figure 28: Memory usage of Youku with and without TrafficGuard.

Efficiently Delivering Online Services over Integrated Infrastructure

Hongqiang Harry Liu *Raajay Viswanathan* *Matt Calder* *Aditya Akella*
Ratul Mahajan *Jitendra Padhye* *Ming Zhang*

Microsoft University of Wisconsin–Madison USC

Abstract— We present Footprint, a system for delivering online services in the "integrated" setting, where the same provider operates multiple elements of the infrastructure (e.g., proxies, data centers, and the wide area network). Such integration can boost system efficiency and performance by finely modulating how traffic enters and traverses the infrastructure. But fully realizing its benefits requires managing complex dynamics of service workloads. For instance, when a group of users are directed to a new proxy, their ongoing sessions continue to arrive at the old proxy, and this load at the old proxy declines gradually. Footprint harnesses such dynamics using a high-fidelity model that is also efficient to solve. Simulations based on a partial deployment of Footprint in Microsoft's infrastructure show that, compared to the current method, it can carry at least 50% more traffic and reduce user delays by at least 30%.

1 Introduction

The emergence of cloud computing is reshaping how online services and content are delivered. Historically, the three types of infrastructure required for service delivery—*i*) data centers (DC) that host application logic and state; *ii*) edge proxies that terminate TCP or HTTP connections and cache content close to users; *iii*) wide area networks (WAN) that connect DCs and proxies—were owned and operated by different organizations (Figure 1a). But now, large cloud providers such as Amazon, Google, and Microsoft operate all three types of infrastructures for their own and their customers' services [6, 7, 9] (Figure 1b). Infrastructure integration is also ongoing for massively-popular service providers such as Facebook as they leverage their scale to amortize infrastructure cost [8], and for large ISPs as they begin offering content distribution services [10].

Infrastructure integration allows one to take a holistic view of the system to improve both performance and efficiency. For instance, the state of the WAN (e.g. residual capacity of proxy-to-DC path) can be factored into deciding which proxies serve which users.

But to our knowledge, current systems for delivering online services do not take such a holistic view. Many such systems were designed for the traditional setting [13, 4, 11, 29]. Even those that operate in integrated settings fail to leverage its unique opportunities. For example, currently, in Microsoft's network [14], WAN traffic engineering (TE) operates independently, with no advance knowledge of load placed on it by edge proxies and has no ability to steer load to a different proxy or DC to relieve hotspots. At the same time, the edge proxies have no knowledge of WAN TE. When they select DCs for a user session, they need to know the quality of the WAN paths to different DCs. They do so by probing the paths, which is akin to looking in the rear view mirror: it can detect congestion only *after* it occurs and cannot guarantee congestion-freedom when load is moved.

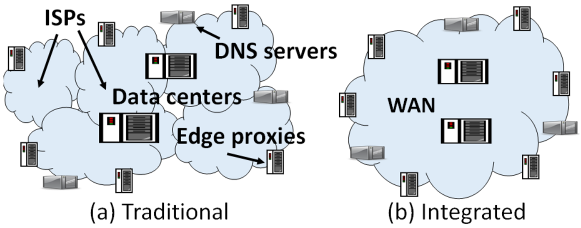


Figure 1: *Online service delivery infrastructures.*

This paper describes Footprint, a system for delivering services that exploits the opportunities offered by the integrated context. Using an SDN-like centralized control model, it jointly coordinates all routing and resource allocation decisions, to achieve desired objectives. It decides how to map users to proxies, proxies to DC(s), and traffic to WAN paths, and configures all components used for service delivery, including network switches, proxies, load balancers, and DNS servers to achieve this mapping.

While it is not surprising that coordination among system components (e.g., joint optimization of WAN TE and proxy load management) can help, we show that fully realizing the potential of infrastructure integration requires faithful modeling of system dynamics. A major issue is that after we change system configuration, its impact is not immediate but manifests only gradually.

The reason is that ongoing user sessions will continue to use the proxy that they picked at session start. Thus, when the controller changes the proxy (or the DC) mapping for a group of users, traffic from those users will not move immediately. Instead, the load on the second proxy (or the second DC) will increase as new sessions arrive and that on the first proxy (or DC) will decrease as old sessions depart. The system model and control algorithms must correctly account for this lag. Traditional network TE controllers such as SWAN [18] and B4 [19] do not have to deal with this lag, since they operate at packet granularity, and the impact of a configuration change is immediate.

In this paper, we illustrate the modeling challenge using data from Microsoft's service-delivery infrastructure, and we devise techniques to address it. To capture temporal variations, we model system load and performance as a function of time. Solving time-based models can be intractable (e.g., time is continuous), but we show how all load and performance constraints can be met by considering a small number of time points. The basic issue tackled by our model—gradual impact of configuration changes—arises in many other systems as well, such as session-based load balancers, middleboxes, and even traditional CDNs. Our model is flexible and can be adapted to improve the efficiency of these systems too.

In addition to the modeling challenge, we address a number of practical issues to design a scalable and robust system. For example, we need to estimate the latency to various edge proxies from different user groups in a scalable manner. We will discuss these issues, and our solutions for them in more detail later in the paper.

We implement our model and other techniques in a Footprint prototype. This prototype is deployed fully in a modest-sized testbed, and its monitoring aspects are deployed in Microsoft's infrastructure. We evaluate Footprint using these deployments and trace-driven simulations. We find that it enables the infrastructure to carry at least 50% more traffic, compared to Microsoft's current method that does not coordinate the selection of proxies, DCs, and WAN paths. At the same, it improves user performance by at least 30%. We also show that Footprint's system model is key to achieving these gains.

2 Background and Motivation

Figure 1 shows a high-level view of online service delivery infrastructure. DCs, which usually number $O(10)$, host application logic and hard state. Users connect to DCs via edge proxies. The proxies help boost performance by terminating TCP and HTTP connections (coming over possibly lossy last mile paths) close to the user and by caching some content (so it does not need to be fetched from a distant DC).

In the traditional architecture, the DCs, the edge proxies and the WAN that connects them are operated by different entities. For example, the DCs may be owned by a hosting service, the edge proxies may be owned by a company like Akamai and various ISPs may provide connectivity between the DCs, and to the edge proxies.

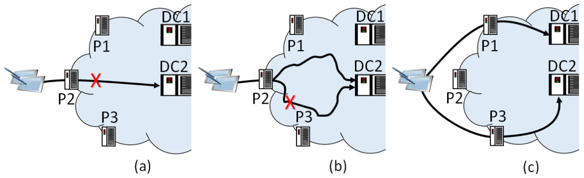


Figure 2: *Spatial modulation via joint coordination. (a) Path between P2 and DC2 is congested. (b) WAN TE alone cannot resolve this congestion because other paths between P2 and DC2 have low available capacity. (c) Congestion is resolved when user-to-proxy mapping and WAN TE are done jointly, moving users to other proxies with uncongested paths to DCs.*

As discussed earlier, large cloud providers like Microsoft and Google, are moving toward a more integrated architecture, where a single entity owns and operates the DCs, the WAN connecting the DCs, and the edge proxies.

Regardless of the architecture, any online service delivery system makes three decisions for user requests: (i) selecting the proxy for the request (ii) selecting the DC(s) for user sessions at a proxy, and (iii) selecting WAN path(s) for traffic between proxies and DCs.

In the traditional setting, the three decisions are made largely independently of one another, and typically without the benefit of global knowledge. A third-party like Akamai makes a decision about which proxy the user selects, and which DC the request will be served from. Various ISP routing policies decide how the traffic flows between the DCs and the proxies.

Even in an integrated online service provider (OSP), these decisions are often made independently. For example, in Microsoft's FastRoute system, anycast routing is used to direct clients to nearby proxies [14]. The proxies independently decide which DCs to route the request to, and the WAN TE is performed independently as well.

In this paper, we argue for making service-delivery decisions jointly. Joint decisions can significantly improve efficiency, not only because of global information, but also by offering new knobs that were previously unavailable. For example, consider Figure 2, where congestion appears between P2 and DC2. In the traditional setting, WAN TE cannot change how traffic enters and exits the network as that is determined by proxy and DC selection. To relieve congestion, it must rearrange how traffic flows within the network. However, sometimes that may not be sufficient (Figure 2b). Joint decisions can "spatially modulate" the traffic (i.e., change where it enters or exits the WAN) by simultaneously controlling the proxy and DC selection. As shown in Figure 2c, such spatial modulation can help relieve congestion.

Spatial modulation is especially helpful when a large fraction of WAN traffic is for user-facing services. This situation holds for large cloud providers; they have a separate WAN for non-user-facing traffic [18, 19]) To

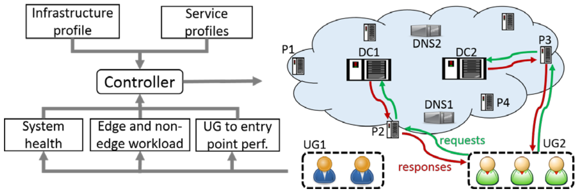


***Figure 3:** Overview of* Footprint.

evaluate the benefit of spatial modulation in practice, we analyze traces from Microsoft's infrastructure, which runs WAN TE and service delivery controller independently [14]. We identified congestion events in the WAN as those where the utilization of at least one link is over 80% during a 5 minute window. We find that all of these events could be resolved using spatial modulation of service traffic. We also repeated the study by artificially scaling traffic by 50%: the number of congestion events went up by 1200% (because our WAN is heavily utilized), but all of them could still be resolved. This advantage of spatial modulation underlies the efficiency and performance improvements of Footprint (§7).

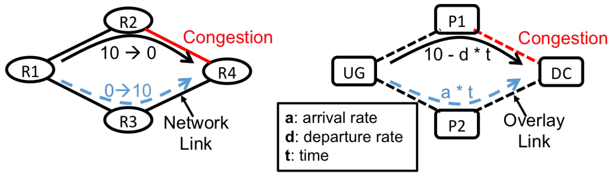
While joint decisions can help, we will see that accurate modeling of system dynamics is necessary to realize its benefits. Next, we provide an overview of the Footprint architecture, and outline key challenges.

3 Overview of Design and Challenges

Figure 3 shows an overview of Footprint. The controller is bootstrapped with infrastructure and service profiles. Infrastructure profile describes the topology, capacity in terms of multiple resources (e.g., CPU, memory, bandwidth), and latency of each component. A service's profile describes which proxies and DCs host it—not all services may be hosted everywhere—and any constraints on mapping users to proxies and DCs (e.g., Chinese users must be served from China). When running, the controller gets up-to-date information on system health, workload, and user-to-proxy delays. Periodically, or after a failure, the controller computes and deploys new system configuration based on this information. This configuration determines, for the next period, how user requests map to proxies, which DCs are used by proxies, and which WAN paths are used.

Our design must address three categories of challenges: obtaining the necessary inputs, computing the desired configuration, and implementing the computed configuration. We provide a brief overview of these challenges in this section. Future sections provide more detail, with a focus on the system model.

Obtaining dynamic inputs: In addition to static inputs such as WAN topology, the controller needs up-to-date information about user-to-proxy delays, the load on the system (i.e. load on WAN links, data center and proxy utilization), and information about system health (e.g. which links or proxies have failed). We have scalable infrastructure in place to collect the needed information about WAN and proxy load and health [30].



(a) Instantaneous configuration change in traditional TE.

(b) Gradual configuration change in session-level overlay routing.

***Figure 4:** Session affinity results in gradual load changes in session routing on top of server overlays.*

A key challenge lies in scalably collecting information about user-to-proxy delays. We address it in two ways. First, we group users into groups—a user group (UG) is a set of users that are expected to have similar relative latencies to edge proxies (e.g., because they are proximate in Internet topology). Second, we measure delays between UGs and proxies in a probabilistic manner. §5 describes these aspects in more detail.

Computing the configuration: We initially believed that we could compute system configurations using a linear program (LP) similar to TE controllers such as SWAN [18]. However, we realized that the Footprint controller faces qualitatively different problems. The key issue is that service sessions last longer than individual packets and these sessions stick to their originally chosen proxies and DCs during their lifetime.

More specifically, online services rely on DNS to direct different users to different proxies—IP addresses of the desired proxies are returned when the user looks up the name of the service. The mapping of name to IP addresses is changed to move load from one proxy to another. The problem is that DNS changes cannot change traffic distribution instantaneously. In addition to DNS mappings being cached at the LDNS servers for the TTL duration, there are two other problems. First, DNS mappings may be cached at the client well beyond the TTL value (e.g., many browsers will not look up the same hostname again within a tab, as long as the tab is open). Second, persistent TCP connections used by HTTP 1.1 and 2.0 can last well beyond DNS TTL as well.

This caching means that even after the Footprint controller updates a DNS mapping to point a UG to a new proxy, the traffic from ongoing sessions from that UG continues to arrive at the old proxy. The proxy must continue to send traffic from ongoing sessions to the same DC. Otherwise, those sessions may be abruptly terminated whenever system configuration is changed (e.g., every 5 minutes).

Session stickiness makes it harder to compute robust system configurations compared to traditional TE. For instance, in Figure 4(a), traffic from R1 to R4, is ini-

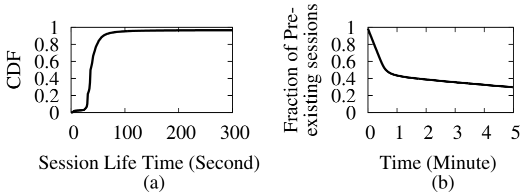


Figure 5: *Session life time.*

tially routed via R2. When the link R2-R4 is congested, TE controller configures R1 to forward the traffic via R3. This change is near instantaneous, and more importantly, largely transparent to the applications. However, the Footprint controller does not have this luxury. Figure 4(b) shows an example. Initially, a group of users (UG) use proxy P1 to access the service S hosted in the data center (DC). When the path P1-DC is congested, we need to reroute the traffic via P2. This can be done by changing the DNS mapping; i.e. the name for service S resolves to the IP address of proxy P2. However this change only affects new user sessions, and traffic from old sessions continues to arrive at P1.

The severity of the problem is illustrated in Figure 5(a). Using data logged across all our proxies, it shows the CDF of the lifetime of TCP connections for the Bing search service. We see that 5% of the connections last longer than 100 seconds. In our current implementation, Footprint adjusts DNS mappings every 5 minutes. Since new HTTP sessions arrive roughly uniformly in a five minute interval, a large fraction TCP connections continue to send data to the "old" proxy after the mapping is updated. Figure 5(b) shows that the number of sessions that are still active as a function of time. Even if the DNS mapping is changed at the end of the 5 minute period, over 20% of the sessions will continue to send data to the previous proxy. The previous proxy must continue to handle this "old" traffic, and send it to the same DC as before.

We deal with this challenge by incorporating session lifetime and workload migration dynamics into our system model, as described in the next section. The reader may wonder why we simply do not kill the old connections, which would obviate the need for modeling temporal behavior. But, as shown above, a large number of "old" connections are active on each epoch boundary. It is unacceptable to kill so many connections every five minutes. We may alleviate the problem by updating system configuration less frequently (e.g., an hour). But we need our system to be responsive and react quickly to demand bursts (e.g., flash crowds) and faster updates lead to greater efficiency [18]. We may also alleviate the problem by updating the client code to gracefully handle changes to proxy mappings. But Footprint must accommodate a large number of already-deployed applications.

Implementing the computed configuration: The computed configuration is implemented by updating DNS mappings, proxy-to-DC mappings, and weights on WAN paths. One issue that we face here is that changing UG-to-proxy mappings (e.g., in response to WAN congestion) can force user traffic onto paths with unknown capacities. While we monitor UG-to-proxy path delays, we are not reliably aware of path capacities. We thus prefer that UGs continue to use current paths to the extent possible. To ensure this, Footprint uses load balancers at network edge that can forward user requests to remote proxies. These load balancers allow us to decouple how users reach our infrastructure and how their traffic flows internally. We omit details due to lack of space.

4 System Model

The Footprint controller periodically decides how requests and responses from each UG are going to traverse the infrastructure. For example, in Figure 3, suppose it makes a fraction of sessions from UG2 go through edge proxy P2 and data center DC1, it also simultaneously computes how to route request traffic in network from P2 to DC1 and response traffic from DC1 to P2.

The controller computes how to assign *new* sessions from each UG to a collection of end-to-end paths (or e2e-paths) which includes two "servers"—an edge proxy y and datacenter c—and two WAN paths—request and response paths between y and c.

Each network path is a pre-configured tunnel, i.e., a series of links from the source to destination switch; there are usually multiple tunnels between a source-destination pair. Once a session is assigned to an e2e-path, it will stick to the assigned proxy and DC; the WAN paths taken by the traffic may change.

The assignments from sessions to e2e-path impacts system efficiency as well as the latency experienced by users. The controller must enable the infrastructure to accommodate as much workload as possible, while ensuring that the proxies, DCs and network paths are not overloaded and that traffic prefers low-latency paths. The key to meeting these goals is to model the load on resources with high fidelity, which we do as described below.

4.1 Preliminaries

Table 1 shows key notations in our model, including its inputs and outputs. The outputs contain routing decisions for two types of traffic. The first type is *unsettled edge traffic* due to new user sessions for services hosted on the edge proxies. Here, the routing decision $\mathbf{w}_{\theta,g}$ denotes the fraction of new sessions from UG g assigned to e2e-path θ. The second type is *settled traffic*, due to existing edge sessions that stick to their servers and all non-edge traffic carried by the WAN. Here, the routing decision $\omega_{p,s,d}$ denotes the fraction of non-edge (i.e., non-service) traffic

Inputs	
g	A user group (UG)
a_g^j	Session arrival rate of g in jth epoch
$q(t)$	CDF of session lifetime
θ	An e2e-path
Θ_g	E2e-paths that can selected by UG g
e	A "server" on an e2e path: i.e. an edge proxy or a datacenter
p, l	p: network path; l: a network link
bw_l	Bandwidth of link l
$P_{s,d}$	All network paths that starts from server s and ends with server d
$\xi'_{s,d}$	Non-edge traffic demand from s to d
$h_{\theta,g}$	Latency experienced by g when going through θ
α	A resource (e.g. CPU, memory etc.) at an edge proxy or a datacenter
$M_{\alpha,e}$	Capacity of resource α at (e)
c_{req}, c_{rsp}	Bandwidth consumption of a request, response
T	Length of an epoch
Intermediate variables	
$\mu_{\alpha,z}(t)$	Resource α's utilization on z
$n_{\theta,g}(t)$	Number of sessions on θ from g
$n_{e,g}(t)$	Number of sessions on e from g: $\sum_{\forall\theta:e\in\theta} n_{\theta,g}(t)$
$a_{\theta,g}(t)$	Session arrival rate on θ from g
$a_{e,g}(t)$	Session arrival rate on e from g: $\sum_{\forall\theta:e\in\theta} a_{\theta,g}(t)$
$\xi_{s,d}(t)$	Traffic of settled sessions s to d
$f(t)$	CCDF of session lifetime
Outputs	
$\mathbf{w}_{\theta,g}$	Weight of new sessions of UG g on θ
$\omega_{p,s,d}$	Weight of traffic from s to d on network path p

Table 1: *Key notations in* Footprint *model.*

from source s to destination d assigned to network path p. Note that s and d represent WAN endpoints connected to datacenters, edge proxies, or neighboring ISPs. For instance, for non-edge traffic s and d may be a neighboring ISP and a datacenter; for service request traffic generated from UGs, s is the proxy, while d is the datacenter.

Constraints: Because the sessions from g can only be assigned to a subset of e2e-paths Θ_g whose proxies are close enough to g, and similarly traffic from s to d can only traverse a subset of network paths $P_{s,d}$ that connect s and d, we have the following constraints on routing:

$$\forall g: \quad \textstyle\sum_{\forall\theta} \mathbf{w}_{\theta,g} = 1, \text{ if } \theta \notin \Theta_g, \text{ then } \mathbf{w}_{\theta,g} = 0 \qquad (1)$$

$$\forall s,d: \quad \textstyle\sum_{\forall p} \omega_{p,s,d} = 1, \text{ if } p \notin P_{s,d}, \text{ then } \omega_{p,s,d} = 0 \qquad (2)$$

Before describing the system model based on which we compute these routing decisions, we list the assumptions made in our modeling.

Assumptions: We assume that a DC is involved in serving each user request. This assumption does not imply that there is no caching or local logic at the proxy; it just means that the request cannot be completely fulfilled by the proxy. All our services require personalized responses based on state that is only maintained in DCs. It is straightforward to modify our model when used with services where this behavior does not hold.

We assume that the session arrival rate for a user group g in j-th epoch a_g^j, is known and fixed. In §5, we describe how arrival rate is estimated. We have empirically verified that the arrival rate is fixed during each epoch, as the epoch length that we use (5 minutes) is short. Our model can be extended to account for errors in the estimated arrival rate [22]. Similarly, we assume that the distribution of session lifetimes, denoted by $q(t)$, is known.

We model proxies and datacenters as monolithic entities, ignoring their internal structure (and hence we refer to them as "servers"). Without this simplifying assumption, the model will become intractable as there will be too many individual servers.

For ease of exposition, we assume that the infrastructure supports only one type of service. This service generates request-response traffic, and the average bandwidths consumed by requests and responses is known (c_{req}, c_{resp}). We define the capacity $M_{\alpha,e}$ of resource α (e.g., CPU, memory) of server e in terms of number of sessions. That is, we say that the CPU on a proxy can handle a certain number of sessions. We assume that this number is known, and fixed for a given α and a given e. Since links can be viewed as a "server" with a single resource—bandwidth—we will occasionally leverage this view to simplify notation. We can extend our model to multiple services and a more detailed view of resource and bandwidth consumption [22].

Finally, we assume the system's objective is to find end-to-end paths that minimize user delays. We do not consider properties such as throughput or loss rate, but we model the impact of high utilized resources on delay.

4.2 Temporal system dynamics

To model resource utilization, we first model the number of active sessions consuming that resource. Let z denote any element of an end-to-end path - a "server" or a link. The number of active sessions on z is:

$$n_z(t) = \sum_{\forall g} \sum_{\forall\theta:e\in\theta} n_{\theta,g}(t) \qquad (3)$$

where, $n_{\theta,g}$(t) is the number of active sessions from UG g on e2e-path θ at time t, and thus n_z(t) is the total number of active sessions on element z. $n_{\theta,g}(t)$ evolves with time, as new sessions arrive and old ones depart.

Consider epoch k, which lasts from time $t \in [kT,(k+1)T]$, where T is the epoch length. At the beginning of the epoch ($t = kT$), there are $n_{\theta,g}^{old}$(kT) pre-existing sessions that will terminate per the pattern defined by the distribution of session life time. Simultaneously, new sessions will continuously arrive, some of which terminate inside the current epoch and others will last beyond the epoch. At any given time, the total number of sessions in the epoch is:

$$n_{\theta,g}(t) = n_{\theta,g}^{new}(t) + n_{\theta,g}^{old}(t) \qquad (4)$$

We must faithfully model how $n_{\theta,g}^{new}(t)$ and $n_{\theta,g}^{old}(t)$ evolve with time to provide high performance and efficiency.

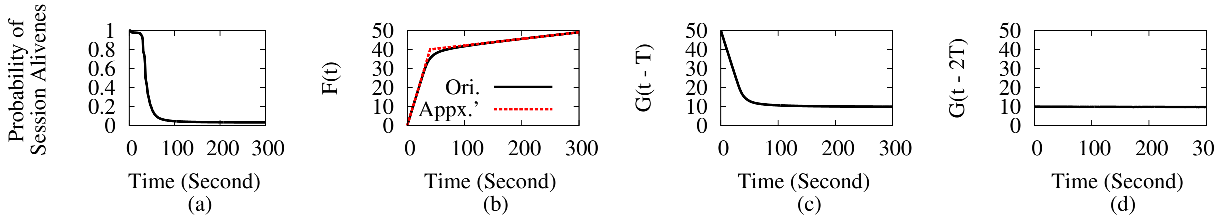


Figure 6: *The pattern functions derived from the session life time distribution of Bing in an epoch. The time (x-axis) on each graph is relative to the start of the epoch.*

New sessions: The new session arrival rate on θ from UG g is:

$$\forall \theta, g : a_{\theta,g} = a_g \times \mathbf{w}_{\theta,g} \tag{5}$$

Recall that a_g is the total arrival rate of sessions from UG g, and we assume it to be fixed within an epoch.

At any given time t within the epoch k, $n^{new}_{\theta,g}(t)$ is the sum of the number of sessions which arrived in interval $[kT,\ t]$ and are still alive at t. From the session life time CDF distribution $q(t')$, we can easily derive $f(t') = 1 - q(t')$, which is probability that a session is still alive after duration t' since it started. Figures 5(a) and 6(a) show examples of $q(t')$ and $f(t')$, respectively.

Therefore, at any given time $\tau \in [kT, t]$, the number of new sessions that arrived in the interval $[\tau,\ \tau + \Delta\tau]$ is $a_{\theta,g} \times \Delta\tau$. Among these sessions, there will be $f(t - \tau)a_{\theta,g} \times \Delta\tau$ sessions left at time t. When $\Delta\tau \to 0$:

$$\begin{aligned} n^{new}_{\theta,g}(t) = \int_{kT}^{t} f(t-\tau) \times a_{\theta,g} d\tau = a_{\theta,g} \times \int_{0}^{t-kT} f(\tau) d\tau \\ = a_{\theta,g} \times F(t - kT) \end{aligned} \tag{6}$$

where $F(t) = \int_0^t f(\tau)d\tau$, which represents the number of sessions alive at t assuming unit arrival rate. Figure 6(b) shows $F(t)$, obtained from Figure 6(a).

Pre-existing sessions: At time t in epoch k, the number of pre-existing sessions that arrived in epoch j ($j < k$) is:

$$\begin{aligned} n^{old,j}_{\theta,g}(t) = \int_{jT}^{(j+1)T} f(t-\tau) \times a^j_{\theta,g} d\tau \\ = a^j_{\theta,g} \times \int_{t-(j+1)T}^{t-jT} f(\tau)d\tau = a^j_{\theta,g} \times G(t - jT) \end{aligned} \tag{7}$$

where $a^j_{\theta,g}$ is the observed arrival rate in epoch j and $G(t) = F(t) - F(t - T)$. Therefore, the total number of pre-existing sessions is:

$$n^{old}_{\theta,g}(t) = \sum_{j=0}^{k-1} a^j_{\theta,g} \times G(t - jT) \tag{8}$$

Figures 6(c) and (d) show examples of $G(t - jT)$ in two epochs prior to current one, i.e., $j = (k-1)T$ and $j = (k-2)T$, respectively when $T = 300$ seconds.

Server utilization: Given the number of active sessions, the utilization of resource α on server e is:

$$\mu_{\alpha,e}(t) = \frac{n_e(t)}{M_{\alpha,e}}, \tag{9}$$

Combining Eqns. 3, 4, 6, 8, and 9, the utilization of a resource α on server e is:

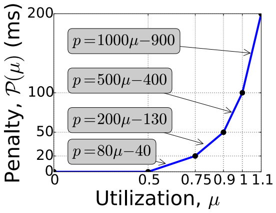


Figure 7: *Penalty function.*

$$\mu_{\alpha,e}(t) = \frac{F(t-kT) \times a_{e,g} + \sum_{j=0}^{k-1} G(t-jT) \times a^j_{e,g}}{M_{\alpha,e}} \tag{10}$$

Link utilization: To model link utilization, we account for non-edge traffic and the fact that requests and responses consume different amounts of bandwidth, c_{req} and c_{rsp}, respectively.

An e2e-path θ contains a request path θ_{req} from UG to DC and a response path θ_{rsp} from DC to UG. Thus, the total edge traffic load generated by new sessions on a network link l is:

$$\mu'_{bw,l}(t) = \frac{\sum_{\forall\theta:l\in\theta_{req}} n^{new}_{\theta}(t)c_{req} + \sum_{\forall\theta:l\in\theta_{rsp}} n^{new}_{\theta}(t)c_{rsp}}{bw_l} \tag{11}$$

Pre-existing sessions stick to their originally assigned servers, but the WAN paths they use can be adjusted. All such sessions from site s to site d generates traffic demand $\xi_{s,d}$:

$$\xi_{s,d}(t) = \sum_{\forall g}[\sum_{\forall\theta:s,d\in\theta_{req}} n^{old}_{\theta,g}(t)c_{req} + \sum_{\forall\theta:s,d\in\theta_{rsp}} n^{old}_{\theta,g}(t)c_{rsp}] \tag{12}$$

Links are shared by edge and non-edge traffic. Let $\xi'_{s,d}$ be the traffic demand from source router s to destination router d, the link load by non-edge traffic on link l is:

$$\mu''_{bw,l}(t) = \frac{\sum_{\forall s,d}\sum_{\forall p:l\in p}[\xi_{s,d}(t) + \xi'_{s,d}] \times \omega_{p,s,d}}{bw_l} \tag{13}$$

Thus, the total utilization of network link l should be:

$$\mu_{bw,l}(t) = \mu'_{bw,l}(t) + \mu''_{bw,l}(t) \tag{14}$$

4.3 Optimization objective

Equations 10 and 14 model the impact of routing decisions on resource utilization. For computing the final decisions, resource utilization is only half of the story. Our goal is not to exclusively minimize utilization, as that can come at the cost of poor performance if user sessions start traversing long paths. Similarly, the goal is not to exclusively select shortest paths, as that may cause overly high utilization that induces delays for users.

To balance the two concerns, as is common in TE systems, we penalize high utilization in proportion to expected delay it imposes [1]. Figure 7 shows the piece-wise linear approximation of the penalty function $\mathtt{P}(\mu_{\alpha,e})$ we use. The results are not sensitive to the exact shape—which can differ across resource types—as long as the function has monotonically non-decreasing slope.

Thus, our objective function is:

$$\text{min.} \sum_{\forall \alpha,e} \int_0^T \mathtt{P}(\mu_{\alpha,e}(t))dt + \sum_{\forall g,\theta} \int_0^T h_{g,\theta} n_{g,\theta}(t)dt \tag{15}$$

The first term integrates utilization penalty over the epoch, and the second term captures path delay. The variable $h_{g,\theta}$ represents the total latency when sessions of UG g traverse e2e-path θ. It is the sum of UG-to-proxy and WAN path delays.

4.4 Solving the model

Minimizing the objective under the constraints above will assign values to our output variables. However, our model uses continuous time and we must ensure that the objective is limited at all possible times. To tractably guarantee that, we make two observations. First, $n^{new}_{\theta,g}(t)$ monotonically increases with time and is also concave. The concavity is valid if only $\frac{dF(t)}{dt} = f(t)$ is monotonically non-increasing with t, which is always true because $f(t)$ is a CCDF (complementary cumulative distribution function). Hence, we can use a piecewise linear concave function $F'(t)$ that closely upper-bounds $F(t)$. For instance, the red, dashed line in Figure 6b shows a two-segment $F'(t)$ we use for Bing.

The second observation is that $n^{old}_{\theta,g}(t)$ is monotonically decreasing and convex, e.g. Figure 5(b). The convexity depends on both the shape of $f(t)$ and the length of epoch T we choose. We found the convexity of $n^{old}_{\theta,g}(t)$ is valid for all the services in our infrastructure. This, in this paper, we assume for simplicity that $n^{old}_{\theta,g}(t)$ is always convex. Otherwise, one can may use a piecewise linear function to upper-bound $n^{old}_{\theta,g}(t)$.

Therefore, when we use $F'(t)$ instead of $F(t)$, from Eqn. 10 we derive:

$$\mu_{\alpha,e}(t) \le \bar{\mu}_{\alpha,e}(t) = \frac{F'(t-kT)a_{e,g} + \sum_{j=0}^{k-1} G(t-jT)a^j_{e,g}}{M_{\alpha,e}} \tag{16}$$

where $\bar{\mu}_{\alpha,e}(t)$ is upper-bounding $\mu_{\alpha,e}(t)$ all the time, so that we can limit $\mu_{\alpha,e}(t)$ by limiting $\bar{\mu}_{\alpha,e}(t)$.

Since $\sum_{j=0}^{k-1} G(t-jT)$ is also convex with time t, and let $\tau_1,\ldots,\tau_m$, where $\tau_1 = 0, \tau_m = T$, be the conjunction points of linear segments in $F'(t)$, $\bar{\mu}_{\alpha,e}(t)$ becomes a piecewise convex function and each convex piece i ($1 \le i \le m-1$) has boundary τ_i and τ_{i+1}. Because the maximum value of a convex function must be on the boundary, the maximum value of convex piece i in $\bar{\mu}_{\alpha,e}(t)$ happens on either $t = \tau_i$ or $t = \tau_{i+1}$. Hence, overall, the maximum value of $\bar{\mu}_{\alpha,e}(t)$ always happens at a collection of particular moments which are $\tau_1,\ldots,\tau_m$. Formally, we have:

$$\bar{\mu}^{max}_{\alpha,e} = \max\{\bar{\mu}_{\alpha,e}(\tau_i) | i = 1,\ldots,m\} \tag{17}$$

Similarly, for link utilizations and number of sessions, we also have:

$$\bar{\mu}^{max}_{bw,l} = \max\{\bar{\mu}_{bw,l}(\tau_i) | i = 1,\ldots,m\} \tag{18}$$

$$\bar{n}^{max}_{\theta,g} = \max\{\bar{n}_{\theta,g}(\tau_i) | i = 1,\ldots,m\} \tag{19}$$

where $\bar{\mu}_{bw,l}(t)$ and $\bar{n}_{\theta,g}(t)$, similar to $\bar{\mu}_{\alpha,e}(t)$, is also derived from replacing $F(t)$ with $F'(t)$ in Eqns. 14 and 3.

Therefore, we can transfer our original objective function Eqn. 15 into following formulation:

$$\text{min.} \sum_{\forall \alpha,e} \mathtt{P}(\bar{\mu}^{max}_{\alpha,e}) \times T + \sum_{\forall g,\theta} h_{g,\theta} \bar{n}^{max}_{g,\theta} \times T \tag{20}$$

We can now obtain an efficiently-solvable LP combining the new objective in Eqn 20 with constraints in Eqns. 1–5 and 16–19. The penalty function $\mathtt{P}(\mu)$ can also be encoded using linear constraints [15].

5 Footprint Design

We now describe the design of Footprint in more detail.

Defining UGs: We start with each /24 IP address prefix as a UG because we find experimentally that such users have similar performance. In the presence of eDNS, where LDNS resolvers report users' IP addresses when querying our (authoritative) DNS servers, this definition of UGs suffices. However, eDNS is not widely deployed and our DNS servers tend to see only resolvers' (not users') addresses. This lack of visibility means that we cannot map /24 prefixes that share LDNS resolvers to entry point(s) independently. Thus, we merge non-eDNS UGs that share LDNS resolvers, using IP-to-LDNS mapping from our entry point performance monitoring method (described below). We find that such mergers hurt a small minority of users; 90% of the time, when two /24 prefixes have the same LDNS, their relative performance to top-3 entry points is similar.

Entry point performance monitoring: We leverage client-side application code to monitor performance of UGs to different entry points. Our measurement method borrows ideas from prior work [24, 5]. After a query finishes, the user requests a URL from current and alternative entry points. It then reports all response times to a measurement server, which allows us to compare entry points head-to-head, without worrying about differences across users (e.g., home network performance).

However, because there can be $O(100)$ entry points, requesting that many URLs will take a long time and place undue burden on users. We thus perform measurements with a small probability and limit each to three requests. Each URL has the form `http://<guid>.try<k>.service.footprint.com/monitor`, where *guid* is a globally unique identifier and $k \in (1..3)$.

What sits behind *monitor* is a service-specific transaction. For a browsing-type service (e.g., search or social networking) it may correspond to downloading its typical Web page; for a video streaming service, large objects may be downloaded. This way, the response time reflects what users of the service experience.

The measurement mechanics are as follows. Because of the GUID, the URL hostname does not exist in DNS caches and each request triggers a lookup at our DNS server. We resolve the name based on the user's UG and k. For k=1, we resolve to the current-best entry point; for k=2, to a randomly selected entry point from the ten next best; and for k=3, to a random selection from the remaining entry points. Each response-time triplet yields the relative performance of the best and two other entry points. Aggregating across triplets and users provides a view of each entry point's performance for each UG.

This view is more up-to-date for better entry points for a UG as they are sampled from a smaller set (of 10). When a UG's entry point is changed, it is likely mapped to another nearby entry point; up-to-date view of such entry points is important, which would be hard to obtain with unbiased sampling of all entry points.

Finally, we learn the mapping from users' IP addresses to LDNS resolvers by using GUIDs to join the logs at HTTP transaction servers (which see users' addresses) and DNS servers (which see resolver addresses).

Clustering UGs: After LDNS-based grouping, we get O(100K) UGs, which poses a scaling problem for our LP solver. To reduce the number of UGs, we aggregate UGs at the start of each epoch. For each UG, we rank all entry points in decreasing order of performance and then combine into virtual UGs (VUG) all UGs that have the same entry points in the top-three positions in the same order. We formulate the model in terms of VUGs. The performance of a VUG to an entry point is the average of the aggregate, weighted by UGs' number of sessions. For our infrastructure, this clustering creates O(1K) VUGs, and we observe only a marginal decline in efficiency ($\approx$3%) due to our inability to map individual UGs.

System workload: The controller estimates the workload for the next epoch using workload information from previous epochs. DNS servers report the arrival rates of new sessions for each UG and each service; proxies report on resource usage and departure rate of sessions; and network switches that face the external world report on non-edge traffic matrix (in bytes/second). Edge workload is captured in terms of all resource(s) that are relevant for allocation (e.g., memory, CPU, traffic). We use exponentially weighted moving average (EWMA) to estimate workload for the next epoch. We also use linear regression to infer per-session resource consumption (e.g., CPU cycles) for each service, using overall resource usage and number of active sessions per service.

System health: When failures occur, health monitoring services at proxy sites and DCs inform the controller how much total site capacity is lost (not which servers). This information granularity suffices because the controller does not allocate sessions to individual servers at a site and relies on local load balancers for that. In contrast, network failures are exposed at link-level, so that the controller can determine network paths.

Because it may take a few seconds for the controller to react to server or link failures (§7.4), instead of waiting for the controller to update the configuration, load balancers and routers react to failures immediately by moving traffic away from failed components. Such movements can cause transient congestion in our current design, which we plan to address in the future using forward fault correction (FFC) [23].

Robustness to controller failures: To make the system robust to controller or hosting-DC failures, we run multiple controllers in three different DCs in different geographic regions. All dynamic information required for the optimization (e.g., system workload) is reported to all controllers in parallel. The controllers elect a unique leader using ZooKeeper [33]. Only the leader computes the new system configuration and updates the infrastructure, which ensures that updated system state is not inconsistent even if different controllers happen to have different views of the current workload (e.g., due to delays in updating information at a given controller). When the leader fails, a new leader is elected. The new leader can immediately start computing new system configurations as it already has all the requisite inputs.

6 Footprint Prototype

We have implemented the Footprint design outlined above. The client-side functionality for entry point performance monitoring is a JavaScript library that can be used with any Web service. This library is invoked after page load completes, so that it does not interfere with user experience. When fetching a URL in JavaScript, we cannot separate DNS lookup and object download times. To circumvent this limitation, before fetching the URL, we fetch a small object from the same hostname. Then, because of DNS caching, the response time of the URL does not include DNS lookup time. In cases where the browser supports the W3C Resource Timing API, we use the precise object fetch time. We implemented the DNS server-side functionality by modifying BIND [3] and proxy functionality using Application Request Routing [2], which works with unmodified Web servers. We use Mosek [26] to solve the LP.

Timely processing of monitoring data is critical. A particularly onerous task is the real-time join between HTTP and DNS data, to know which endpoints our JavaScript has measured and to attach detailed network and geographic information to each measurement. To help scale, we build our processing pipeline on top of Microsoft Azure Event Hub and Stream Analytics.

To scale the computation of new configurations, we limit the number of e2e-paths that a VUG can use. Specifically, we limit each VUG to its best three entry points—the ones on which VUG was clustered—each load balancer to three proxies, and each source-destination switch pair to six paths (tunnels) in the WAN. In our benchmarks, these limits speed computation by multiple orders of magnitude, without noticeably impacting system efficiency or performance.

We deployed a prototype of Footprint in a modest-sized testbed. This environment emulates a WAN with eight switches and 14 links, three proxy sites, and two DCs. Proxy sites and DCs have one server each. We have 32 PCs that act as UGs and repeatedly query a service hosted in the DC. UG to entry point delays are controlled using a network emulator.

The monitoring aspects of Footprint, but not the control functionality, are also deployed in Microsoft's service delivery infrastructure. This allow us to collect data from $O(100)$ routers, $O(50)$ edge sites, and $O(10)$ DCs worldwide. The JavaScript library is randomly included in 20% of Bing user requests. We use the data from this deployment to drive simulations to evaluate Footprint.

7 Experimental Evaluation

We evaluate Footprint along several dimensions of interest. First, we use the testbed to show the viability and value of jointly controlling all types of infrastructure components. It is not intended to shed light on efficiency and performance of Footprint in a real deployment. To assess those aspects, we conduct large-scale simulations based on data gathered using the monitoring deployment of Footprint in our production environment.

7.1 Testbed results

We use a simple experiment on our testbed to demonstrate the value of spatial traffic modulation. In this experiment, we recreate the example in Figure 2. Recall that in this example the WAN gets congested such that no path between the entry point P2 and DC2 is congestion-free. We create such congestion by injecting non-edge traffic that uses those paths.

Figure 8 shows the results. It plots the response time for UGs that are originally mapped to P2 and DC2, with Footprint and with WAN TE alone. WAN TE estimates the WAN traffic matrix based on recent history and routes traffic to minimize link utilization while using short paths [18]. We see that soon after congestion occurs, Footprint spatially modulates the traffic such that UGs' performance is restored. But WAN TE is unable to resolve congestion and performance issues as it cannot change UGs' proxy and DC selections.

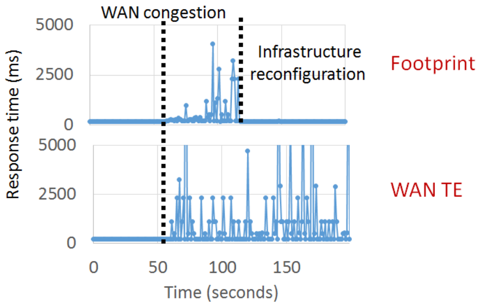


Figure 8: *Testbed experiment: WAN congestion.*

7.2 Efficiency and performance

To understand the efficiency and performance of Footprint at scale, we conduct detailed simulations using data from Microsoft's service delivery infrastructure. Our simulations use a custom, fluid-level simulator.

7.2.1 Methodology and data

To understand the benefit of Footprint's joint optimization, we compare it to a system similar to Microsoft's current approach, where *i*) anycast routing is used to map UGs to their best proxy; *ii*) each edge proxy independently chooses the closest DC for its user sessions based on up-to-date delay measurements; and *iii*) WAN TE periodically configures network paths based on observed traffic, to minimize maximum link utilization [18]. In our simulations, the control loops, for DC selection at each proxy and for WAN TE, run independently every 5 minutes. To mimic anycast routing, we use our monitoring data to map UGs to the best proxy, which enables a fair comparison by factoring out any anycast suboptimality [5]. We also assume that proxies are not the bottleneck, to remove the impact of anycast routing's inability to evenly balance load across proxies, which a different user-to-proxy mapping system may be able to achieve. Abusing terminology, we call this system FastRoute, even though the FastRoute paper [14] discusses only user-to-proxy mapping and not WAN TE.

We drive simulations using the following data: *i*) timestamps of new sessions obtained from system logs; *ii*) distribution of session lifetimes; *iii*) UG to entry point performance data from our monitoring deployment; *iv*) propagation latencies and capacities of all links in the WAN; *v*) server capacities at the edge proxies and data centers; *vi*) non-edge traffic carried by the WAN; and *vii*)

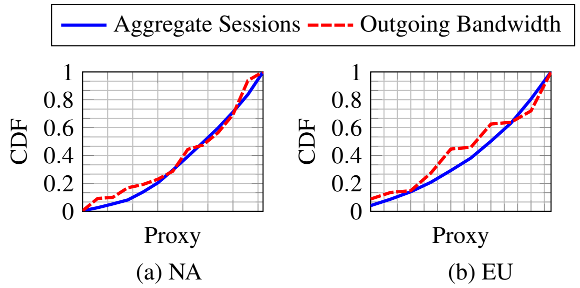


Figure 9: *Distribution of bandwidth and sessions across proxies. User sessions are mapped to the closest proxy. Bandwidth per proxy is measured as aggregate bandwidth of all directly attached links.*

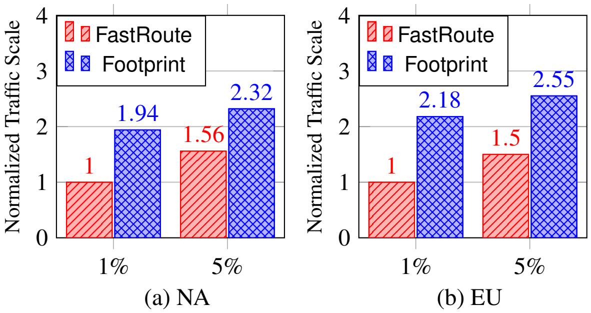


Figure 10: *Efficiency of* FastRoute *and* Footprint *for SLO1 (excess traffic on overloaded components).*

per-session resource consumption (e.g., CPU) estimated using linear regression over the number active sessions.

We show results for North America (NA) and Europe (EU) separately. The infrastructure in the two continents differs in the numbers of proxies, DCs, and the richness of network connectivity. The NA infrastructure is bigger by about a factor of two. The results below are based on one week's worth of data from August 2015. Results from other weeks are qualitatively similar.

To provide a sense of system workload, Figure 9 shows the distribution of closest user sessions and network bandwidth across proxies. Since proxies are not bottlenecks in our experiments, network congestion is a key determiner of performance. While it can occur in the middle of the network as well, congestion occurs more often on links close to the proxies because fewer routing alternatives are available in those cases. We see that, in aggregate, network bandwidth and user sessions of proxies are balanced; more bandwidth is available behind proxies that receive more sessions.

7.2.2 Efficiency

We quantify efficiency of a service-delivery system using *congestion-free scale*—maximum demand that it can carry without causing unacceptable congestion that violates service-level objectives (SLOs). We consider two definitions of unacceptable congestion: *i*) SLO1: across all epochs, the amount of traffic in excess of component capacities should be less than a threshold; *ii*) SLO2: across all epochs, the total traffic traversing overloaded (i.e., load greater than capacity) components should be less than a threshold. The difference in the two SLOs is that when traffic traverses an overloaded component, SLO1 considers only the fraction in excess of the capacity, but SLO2 considers all traffic passing through it. We study multiple congestion thresholds and compute congestion-free scale by iterating over demands that are proportionally scaled versions of the original demand.

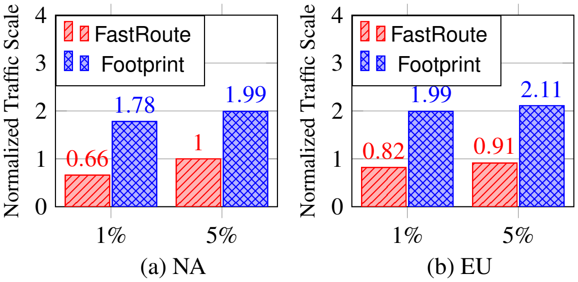


Figure 11: *Efficiency of* FastRoute *and* Footprint *for SLO2 (total traffic on overloaded components).*

Figure 10 shows the congestion-free scale for FastRoute and Footprint with SLO1 for two different congestion thresholds. For confidentiality, we report all traffic scales relative to the congestion-free scale of FastRoute with SLO1 at 1% threshold. We see that Footprint carries at least 93% more traffic when the congestion threshold is 1% and 50% more traffic when the threshold is 5%.

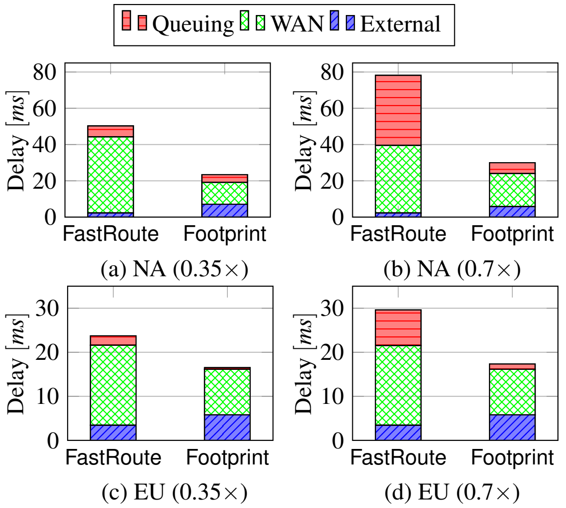
These efficiency gains can be understood with respect to the spatial modulation enabled by joint coordination in Footprint. While on average the load on the proxy is proportional to its network bandwidth (Figure 9), at different times of the day, different regions are active and get congested. By making joint decisions, Footprint can more easily divert traffic from currently active proxies to those in other regions.

Figure 11 shows that Footprint's efficiency gains hold for SLO2 as well, which considers total traffic traversing overloaded components. For 1% and 5% congestion thresholds, Footprint can carry, respectively, 170% and 99% more traffic than FastRoute.

7.2.3 Performance

We quantify performance of user sessions using end-to-end path delays. We study its contributing factors: external (UG-to-proxy) delay, propagation delay inside the WAN, and queuing-induced delays. Queuing delay is quantified using utilization, per the curve in Figure 7. Figure 12 shows the performance of the two system for traffic scales that correspond to 35% and 70% of the congestion-free scale of FastRoute for SLO1. Each bar stacks from bottom three factors in the order listed above.

We see that even when the traffic demand is low (35%), Footprint has 46% (for NA) lower delay. At this



(a) NA (0.35×)

(b) NA (0.7×)

(c) EU (0.35×)

(d) EU (0.7×)

Figure 12: *Delays in the two systems.*

scale, the infrastructure is largely under-utilized. The delay reduction of Footprint stems from its end-to-end perspective. In contrast, FastRoute picks the best proxy for a UG and the best DC for the proxy. The combination of the two might not represent the best e2e path. Such a path may be composed of a suboptimal UG-to-proxy path but a much shorter WAN path. This effect can be seen in the graph, where the external delays are slightly higher but the sum of external and WAN delay is lower.

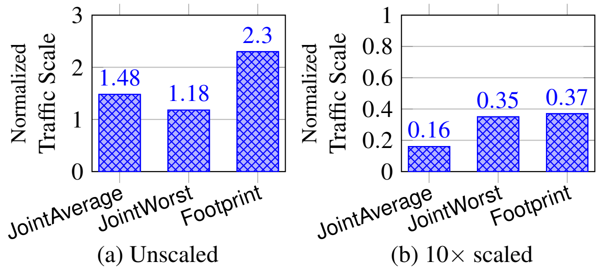
When the traffic demand is high (70%), both systems have higher delay. For FastRoute, most of the additional delay stems from queuing as traffic experiences highly utilized resources. Footprint is able to reduce queuing delay by being better able to find uncongested (albeit longer) paths. Overall, the end-to-end delay of Footprint is at least 30% lower than FastRoute.

7.3 Impact of system model

To isolate the impact of the system model of Footprint, we compare it to two alternatives that also do joint optimization but without the detailed temporal modeling of workload. The efficiency of these alternatives also represents a bound on what existing coordination schemes [16, 20, 28, 12] can achieve when extended to our setting of jointly determining the proxy, WAN path, and DC mappings for user sessions.

- JointAverage Instead of modeling temporal dynamics, based on session lifetimes, JointAverage uses Little's law [21] to estimate the number of active sessions as a function of session arrival rate. If the session arrival rate at a proxy is A per second and the average session lifetime is 10 seconds, on average the proxy will have $10 \times A$ active sessions. These estimates are plugged into an LP that determines how new sessions are mapped to proxies and DCs and how traffic is forwarded in the WAN.

- JointWorst To account for session dynamics, JointWorst makes a conservative, worst-case assumption about load on infrastructure components. Specifically, it assumes that new sessions arrive before any old sessions depart in an epoch. Since we do not do admission control, it is not the case that traffic that is estimated, per this model, to overload the infrastructure is rejected. Instead, the optimization spreads traffic to minimize utilization that is predicted by this model. This model will do well if it overestimates the traffic on each component by a similar amount.



(a) Unscaled

(b) 10× scaled

Figure 13: *Efficiency of different system models for different average session lifetimes on NA infrastructure.*

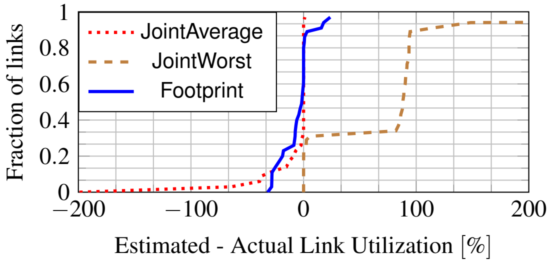


Figure 14: *Fidelity of modeling. CDF of modeled and observed link utilization.*

For NA infrastructure, figure 13(a) compares these two models with Footprint using SLO1 at 5% congestion threshold—the configuration for which Footprint had least gain over FastRoute. We see that Footprint is substantially more efficient. It carries 56% and 96% more traffic than JointAverage and JointWorst.

We find that the gains of Footprint actually stem from its ability to better model load that will be placed on different infrastructure components. To demonstrate it, Figure 14 plots the distribution of estimated minus actual utilization for WAN links for each model. We see that JointAverage tends to underestimate utilization and JointWorst tends to overestimate it. With respect to appropriately spreading load through the infrastructure, neither over- nor under-estimation is helpful.

We also find that, if sessions were much longer, JointWorst performs better because its conservative assumption about existing sessions continuing to load the infrastructure becomes truer. On the other hand, JointAverage gets worse because it ignores the impact of existing sessions altogether, which hurts more when sessions are longer. This is illustrated in Figure 13(b), which shows

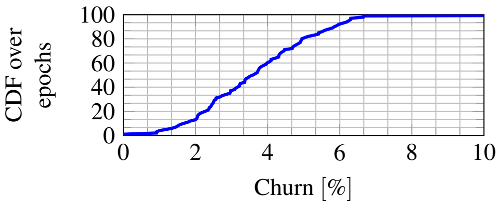


Figure 15: Churn in UG to proxy performance.

the impact on efficiency with average session lifetime multiplied by 10. Because of its modeling fidelity, the benefit of Footprint is not dependent on session lifetime, however, and it is able to provide gains even for these abnormally long sessions.

7.4 Computation time

We measured the time Footprint controller takes to compute system configurations, which includes converting inputs to an LP, solving it, and converting the output to system variables. On an Intel Xeon CPU (E5-1620, 3.70GHz) with 16 GB RAM and using Mosek v7, this time for NA and EU infrastructure is 5 and 0.6 seconds respectively. This level of performance is acceptable given that epochs are much longer (5 minutes). Without clustering of UGs, the running time was greater than 20 minutes for both NA and EU.

7.5 Preliminary experience

We make two observations based on the deployment of Footprint's monitoring components in Microsoft's infrastructure. First, we quantify the fraction of UGs for which the best proxy changes across epochs. If this fraction is substantial, optimal user-to-proxy mapping would move large amounts of WAN traffic, which is better done in coordination with WAN-TE, rather than independently.

Figure 15 shows the fraction of UGs, weighed by their demand, for which the best proxy changes across epochs. On average, this fraction is 5%. It means that a user-to-proxy mapping control loop, operating independently, could move this high a fraction of traffic on the WAN. Joint coordination helps make such movements safely. (In Footprint, since we consider WAN-internal capacity and performance as well, the traffic moved is lower, under 1% in our simulations.)

Second, an unexpected advantage of Footprint's continuous monitoring is that we can discover and circumvent issues in Internet routing that hurt user performance. We have found several such events. In one case, users in the middle of the NA started experiencing over 130 ms round trip delay to a proxy on the west coast, while the historical delay was under 50 ms. In another case, the difference in the delay to reach two nearby proxies in Australia, was over 80 ms. Debugging and fixing such issues requires manual effort, but Footprint can automatically restore user performance in the meanwhile. Anycast-based systems such as FastRoute cannot do that.

8 Related Work

Our work builds on two themes of prior work.

Content distribution systems: Content and service delivery has been an important problem in the Internet for almost two decades. Akamai [25] developed the first large-scale solution, and we borrow several of its concepts such as using edge proxies to accelerate performance and mapping users to proxies based on path performance and proxy load. Since then, researchers have developed sophisticated techniques to tackle this general problem known as replica selection [13, 4, 11, 29]. Solutions tailored to specific workloads (e.g., video) have also been developed [27, 17].

Most of these works target the traditional context in which the WAN is operated separately from the proxy infrastructure. We target the increasingly common integrated infrastructure context, which provides a new opportunity to jointly coordinate routing and resource allocation decisions.

Coordinating decisions: Other researchers have noted the downside of independent decisions for network routing and content distribution. Several works [16, 20, 28, 12] consider coordinating ISP routing and DC selection through limited information sharing; PECAN develops techniques to coordinate proxy selection and external (not WAN) paths between users and proxies [31]; ENTACT balances performance and the cost of transit traffic for an online service provide [32].

Our work differs from these efforts in two ways. First, it includes the full complement of jointly selecting proxies, DCs, and network paths. But more importantly, prior works ignore workload dynamics that arise from session stickiness. Consequently, the best case result of applying their techniques to our setting will approach the JointWorst or JointAverage scheme (§7.3) because, modulo session stickiness, these two schemes optimally map workload to infrastructure elements. We showed that, because it accounts for workload dynamics, Footprint outperforms these schemes. Extending information-sharing techniques to account for such workload dynamics is an interesting avenue for future work.

9 Conclusions

Our work pushes SDN-style centralized control to infrastructure elements beyond networking devices. In doing so, we found that, to maximize efficiency and performance, we must handle complex workload dynamics that stem from application behaviors. This challenge will likely emerge in other systems that similarly push the limits of SDN, and the approach we take in Footprint may inform the design of those systems as well.

Acknowledgments We thank the Microsoft engineers who helped with data acquisition and prototype deployment, including Sreenivas Addagatla, Sorabh Gandhi, Daniel Gicklhorn, Nick Holt, Jon Mitchell, David Schmidt, Prasad Tammana, and Ed Thayer. We also thank the NSDI reviewers and our shepherd, Laurent Vanbever, for feedback that helped improve the paper. Raajay and Aditya were supported in part by National Science Foundation (grants CNS-1302041, CNS-1330308 and CNS-1345249) and the Wisconsin Institue of Software-Defined Datacenters of Madison.

References

[1] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *SIGCOMM*, 2004.

[2] Application request routing. `http://en.wikipedia.org/wiki/Application_Request_Routing`.

[3] BIND. `https://www.isc.org/downloads/bind/`.

[4] S. Buchholz and T. Buchholz. Replica placement in adaptive content distribution networks. In *ACM symposium on Applied computing*, 2004.

[5] M. Calder, E. Katz-Bassett, R. Mahajan, and J. Padhye. Analyzing the Performance of an Anycast CDN. In *IMC*, 2015.

[6] Amazon CloudFront. `http://aws.amazon.com/cloudfront/`.

[7] Windows Azure CDN. `http://azure.microsoft.com/en-us/services/cdn/`.

[8] Facebook CDN. `https://gigaom.com/2012/06/21/like-netflix-facebook-is-planning-its-own-cdn/`.

[9] Google CDN. `https://cloud.google.com/storage/`.

[10] Level3 CDN. `http://www.level3.com/en/products/content-delivery-network/`.

[11] Y. Chen, R. H. Katz, and J. D. Kubiatowicz. Dynamic replica placement for scalable content delivery. In *Peer-to-peer systems*, 2002.

[12] D. DiPalantino and R. Johari. Traffic engineering vs. content distribution: A game theoretic perspective. In *INFOCOM*, 2009.

[13] J. Elzinga and D. W. Hearn. Geometrical solutions for some minimax location problems. *Transportation Science*, 6(4):379–394, 1972.

[14] A. Flavel, P. Mani, D. Maltz, N. Holt, J. Liu, Y. Chen, and O. Surmachev. Fastroute: A scalable load-aware anycast routing architecture for modern cdns. In *NSDI*, 2015.

[15] B. Fortz, J. Rexford, and M. Thorup. Traffic engineering with traditional IP routing protocols. *Comm. Mag.*, 40(10), 2002.

[16] B. Frank, I. Poese, Y. Lin, G. Smaragdakis, A. Feldmann, B. Maggs, J. Rake, S. Uhlig, and R. Weber. Pushing CDN-ISP collaboration to the limit. *SIGCOMM CCR*, 43(3), 2013.

[17] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-scale control plane for video quality optimization. In *NSDI*, 2015.

[18] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[19] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[20] W. Jiang, R. Zhang-Shen, J. Rexford, and M. Chiang. Cooperative content distribution and traffic engineering in an ISP network. In *SIGMETRICS*, 2009.

[21] Little's law. `https://en.wikipedia.org/wiki/Little's_law`.

[22] H. Liu, R. Viswanathan, M. Calder, A. Akella, R. Mahajan, J. Padhye, and M. Zhang. Efficiently delivering online services over integrated infrastructure. Technical Report MSR-TR-2015-73, 2015.

[23] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 527–538, New York, NY, USA, 2014. ACM.

[24] R. Maheshwari. How LinkedIn used PoPs and RUM to make dynamic content download 25% faster. `https://engineering.linkedin.com/performance/how-linkedin-used-pops-and-rum-make-dynamic-content-download-25-faster`, June 2014.

[25] Cloud computing services and content distribution network (CDN) provider — akamai. https://www.akamai.com.

[26] Mosek ApS. https://mosek.com/.

[27] M. K. Mukerjee, D. Naylor, J. Jiang, D. Han, S. Seshan, and H. Zhang. Practical, real-time centralized control for cdn-based live video delivery. In *SIGCOMM*, 2015.

[28] S. Narayana, J. W. Jiang, J. Rexford, and M. Chiang. To coordinate or not to coordinate? wide-area traffic management for data centers. In *CoNEXT*, 2012.

[29] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *INFOCOM*, 2001.

[30] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A network-state management service. In *SIGCOMM*, 2014.

[31] V. Valancius, B. Ravi, N. Feamster, and A. C. Snoeren. Quantifying the benefits of joint content and network routing. In *SIGMETRICS*, 2013.

[32] Z. Zhang, M. Zhang, A. Greenberg, Y. C. Hu, R. Mahajan, and B. Christian. Optimizing cost and performance in online service provider networks. In *NSDI*, 2010.

[33] Apache zookeeper. https://zookeeper.apache.org/.

Scalable and private media consumption with Popcorn

Trinabh Gupta[*†] Natacha Crooks[*‡] Whitney Mulhern[†] Srinath Setty[§] Lorenzo Alvisi[*] Michael Walfish[†]

[*]UT Austin [†]NYU [‡]MPI-SWS [§]Microsoft Research

Abstract. We describe the design, implementation, and evaluation of *Popcorn*, a media delivery system that hides clients' consumption (even from the content distributor). Popcorn relies on a powerful cryptographic primitive: private information retrieval (PIR). With novel refinements that leverage the properties of PIR protocols and media streaming, Popcorn scales to the size of Netflix's library (8000 movies) and respects current controls on media dissemination. The dollar cost to serve a media object in Popcorn is 3.87× that of a non-private system.

1 Introduction and motivation

This paper describes a Netflix-like media delivery system, Popcorn, that provably hides *what* is consumed by its users, at scale and at low (dollar) cost.

Popcorn is motivated by a fundamental tension in the ecosystem of online media consumption. In one camp are people deeply uncomfortable with exposing their media diet, in particular to a centralized media server that can be targeted by either hacking or subpoena. They argue that, philosophically, freedom requires the ability to consume privately [91] and that, practically, access to a person's consumption profile can reveal the person's sexual orientation, political leanings, cultural affiliations, etc. [77, 78, 93].[1] And although many people may in fact *want* to expose their consumption to gain recommendations, there may still be objects that they want to consume without others' knowledge. Another camp counters that media often exists within a commercial framework, and that people who create it and services that distribute it need to be compensated to sustain the ecosystem.

Our work advances a new design point in the realm of private media consumption. Specifically, this paper asks the question, *Is it possible to build a system that hides content consumption while respecting current commercial arrangements, and if so, what would that system cost?*

No answer is likely to apply to all media delivery systems, as they differ widely. YouTube's library, for instance, is large, continuously updated, freely distributed, and supported by advertising. Netflix's library is comparatively small, updated infrequently [6], subject to strict content protection, and supported by paid subscriptions. This paper explicitly targets Netflix-like systems, and adopts the following requirements:

1. *Hide requests comprehensively and provably.* We want to hide consumption from both a network eavesdropper [7, 44] and the content distributor, and avoid the risk of heuristic solutions [18].
2. *Make it affordable even at scale.* Our system should dispense privacy at an attractive price point. The cost should be within a small multiple of what customers pay to access content today.
3. *Respect current controls on content dissemination.* Our solution must be compatible with the existing commercial, legal, and policy regime (copyright, controls on content dissemination, etc.) so as not to fundamentally reorient digital rights.

At first blush, Tor [39] and other anonymity systems [69] (which conceal *who* consumes content) satisfy the above requirements. However, these solutions conflict with commercial media delivery: now Netflix would have to rely on the altruism of Tor nodes. Moreover, the capacity, latency, and reliability on a Tor network is unlikely to match the requirements of Netflix.

Thus, Popcorn turns to a large body of cryptographic protocols known as *Private Information Retrieval*, or *PIR* (§2.2). These protocols [29, 45, 66, 81, 106] allow clients (content consumers) to request content from servers (content distributors) without the servers being able to infer which items the clients requested.

Applying these protocols, however, raises several challenges (§3): the linear overhead of PIR (to respond to a request, the server must compute over its entire library, or else it would learn what the client was *not* interested in); the strict deadlines of media delivery; variable object sizes (PIR assumes all objects are the same size); and a tension surrounding PIR protocol choice (one type of PIR, called CPIR [66], needs only one server, but the overhead is high; another, called ITPIR [29], involves lightweight operations but demands non-colluding servers and hence separate administrative domains, which, among other things, threatens content protection). There is a large and inspiring body of work (§7) addressing some of these issues [12, 13, 25, 31, 33, 35–37, 43, 49, 50, 54, 55, 57, 71, 73, 75, 83, 101, 103, 107], but prior implementations suitable for media delivery at the scale we target levy prohibitive demands on I/O and CPU resources.

Popcorn eases these demands substantially. It provably hides media consumption, scales to the size of Netflix, and respects current controls on media dissemination—

[1] To be clear, we are not challenging the trustworthiness of commercial media services. The issue is that collecting the information in the first place creates the risk of exposure.

with resource overhead that translates to a manageable dollar cost. To do so, Popcorn cherry-picks techniques from the literature on PIR and media on demand, and works through the "systems" ramifications of tailoring them to the context at hand.

Three techniques are central to Popcorn's design. First, Popcorn combines both types of PIR. Media objects, encrypted for content protection, are stored at multiple servers from distinct administrative domains and retrieved using the lighter-weight ITPIR. The much smaller cryptographic keys needed to decrypt those objects are stored at a single server and retrieved using the heavier-weight CPIR. Second, Popcorn amortizes the cost of PIR by batching requests from the large number of concurrent users retrieving content at any given time; by leveraging the properties of media streaming, Popcorn forms large batches without introducing playback delays or interruptions. Third, Popcorn exploits the ability to encode a media object in multiple ways (e.g., by changing its bitrate) to meet the fixed-size-object requirement of PIR.

We experimentally evaluate Popcorn for a Netflix-like workload (10,000 concurrent clients, each streaming different content at 4 Mbps from a library of 8192 movies [1] with an average length of 90 minutes). Popcorn's overheads are high when compared to a non-private baseline: for each request, Popcorn consumes 1080× more computational resources, about 14× more I/O bandwidth, and 2× longer network transfers. However, since CPU is cheap and Popcorn is engineered to conserve the more expensive resources (I/O and network), these overheads, when translated to dollars, are manageable: Popcorn's per-request cost, in terms of dollars, is 3.87× that of the baseline.

Though promising, Popcorn has several limitations (§8). It requires non-colluding servers. Its overheads grow with the library size; this precludes scaling to media libraries that have more than a few tens of thousands of media files (YouTube, for example, has millions [28]). It does not support forward seeking. In addition, the current prototype lacks features that would be required in a full-fledged deployment: online library updates, deployment via CDNs, elasticity, adaptive streaming, royalty payments, and advertising and recommendations. Some of these have natural solutions; others require research.

2 Setting and background on PIR

2.1 Scenario and threat model

The media delivery ecosystem has three principals: a *content creator*, a *content distributor*, and a *content consumer*. The creator (e.g., a movie studio), delegates to the distributor (e.g., an online streaming service like Netflix) the tasks of disseminating content and charging consumers.

We model the content kept by the distributor as a collection L of n objects; we call L the *library*. We assume that a mapping, between the integers $1,\ldots,n$ and the names of the objects in L, is known to the distributor and the consumers. Therefore, a consumer can select a specific object by supplying the corresponding integer.

Threat model. We consider an attacker (for example, the content distributor or a network eavesdropper) trying to infer what object the consumer is accessing. The attacker has full access to the network and to the content of the consumers' requests, but for two restrictions. First, we do not consider side-channel attacks that, for example, use knowledge of where individual consumers pause playback, or of their concurrent web browsing activity. Second, we assume the existence of two non-colluding servers that the distributor can use to serve content. To satisfy this assumption in practice, one can pick servers from separate administrative domains (e.g., from different CDNs [11]). We discuss this topic further in Section 8.

We assume, as do today's media delivery systems [15, 40], that the client-side media decode and display environment can defeat content consumers intent on copying and redistributing content beyond what the distributor allows.

Finally, we treat content integrity as an orthogonal problem that undermines correctness (§2.2) but not privacy. The literature offers standard solutions to guarantee content integrity (content hashing, etc.).

2.2 Private Information Retrieval (PIR)

The high-level goal of PIR protocols aligns with that of Popcorn: they allow a client to use an integer between 1 and n to retrieve any object from a library L of n ℓ-bit objects kept by a set of k servers ($k \geq 1$) without leaking to the servers any information about which object was retrieved. A PIR protocol is structured around three procedures: Query, Answer, and Decode. To privately retrieve object $O_b = L[b]$, the client invokes $\mathsf{Query}(b)$ to produce k query vectors $q_1,\ldots,q_k$, one for each server, and forwards q_j to server S_j ($1 \leq j \leq k$). Each S_j replies with $a_j = \mathsf{Answer}(q_j, L)$. Finally, the client computes $O_b = \mathsf{Decode}(a_1,\ldots,a_k)$ by applying the decode algorithm to the servers' responses.

We want three properties from a PIR protocol:

- ***Correctness.*** If a client requests the object in library L with index b, then the protocol indeed provides it with object $L[b]$.
- ***Privacy.*** After the server sees a query vector, its probability of guessing the client's requested index is no better than if the server had not seen the query in the first place. This property can be generalized to coalitions of $t < k$ servers, requiring that any t out of k servers jointly do not learn any information about the index of the requested object.
- ***Communication efficiency.*** The size of a server's reply must not be much larger than ℓ, and the size of a client's request must be far smaller than ℓ (though

```
Query (index $b$):
    for $i = 1$ to $n$ do
        $f \leftarrow (i == b)$ ? 1 : 0
        $c_i \leftarrow \mathsf{Enc}(pk, f)$
    return $q = (pk, c_1, \ldots, c_n)$
Answer (query vector $q$, library $L$):
    // Represent $L$ as a matrix of $y$-bit integers:
    // $L \in (\{0,1\}^y)^{n \times (\ell/y)}$
    for $j = 1$ to $\ell/y$ do
        $r_j \leftarrow \prod_{i=1}^{n} {c_i}^{L_{i,j}}$
    return $a = (r_1, \ldots, r_{\ell/y})$
Decode (answer $a$, secret key $sk$):
    return $\mathsf{Dec}(sk, r_1), \ldots, \mathsf{Dec}(sk, r_{\ell/y})$
```

Figure 1—A computational PIR (CPIR) protocol based on an additively homomorphic cryptosystem (Gen, Enc, Dec) and due to Stern [98]. (pk, sk) is a (public, private) key pair generated using Gen. n is the number of objects in the library L, and ℓ is the length of each object.

it is acceptable if there is some overhead above the minimum query size of $\log_2 n$ bits).

We discuss below two such PIR protocols.

2.3 Computational PIR (CPIR) protocols

CPIR protocols [66] require only a single, computationally bound server ($k = 1$). They are commonly constructed using additively (not fully [46]) homomorphic public key cryptosystems. A cryptosystem is *additively homomorphic* if $\mathsf{Dec}(sk, \mathsf{Enc}(pk, m_1) \cdot \mathsf{Enc}(pk, m_2)) = m_1 + m_2$, where m_1, m_2 are plaintext messages, $+$ represents addition of two plaintext messages, $\cdot$ is a binary operation (for example, addition, multiplication, etc.) on the ciphertexts, (pk, sk) is a (public, private) key pair generated using the key generation algorithm Gen, Dec is the decryption algorithm, and Enc is the encryption algorithm. Note that Enc is randomized; thus, repeatedly encrypting the same plaintext produces different ciphertexts. Examples of cryptosystems used in CPIR are the Paillier [82] and the lattice-based Ring-LWE [20].

Figure 1 depicts a CPIR protocol, due to Stern [98], that meets the three properties (§2.2):

- ***Correctness.*** $\mathsf{Dec}(sk, r_j) = \mathsf{Dec}(sk, \prod_{i=1}^{n} {c_i}^{L_{i,j}})$, which equals $\sum_{i=1}^{n} \mathsf{Dec}(sk, c_i) \cdot L_{i,j}$ after the application of the additively homomorphic property. But $\forall i \in \{1, \ldots, n\} \setminus b$, $\mathsf{Dec}(sk, c_i) = 0$, by construction of c_i. Similarly, $\mathsf{Dec}(sk, c_b) = 1$. Therefore, $\mathsf{Dec}(sk, r_j) = \mathsf{Dec}(sk, c_b) \cdot L_{b,j} = L_{b,j}$.
- ***Privacy.*** The guarantee that server S does not learn b hinges on S being computationally bounded. All S sees is $q = (pk, c_1, \ldots, c_n)$. If S could systematically guess b (that is, guess which ciphertext is $c_b = \mathsf{Enc}(pk, 1)$), then S could likewise guess which entry is the encryption of 1 (versus 0)—which would contradict the properties of the underlying encryption scheme.

```
Query (index $b$):
    // Generate the first $k-1$ query vectors randomly
    for $j = 1$ to $k-1$ do
        select $q_j \in_R \{0,1\}^n$
    $e_b \leftarrow$ an $n$-bit string with all zeros except at $b$-th position
    $q_k \leftarrow e_b \oplus q_1 \oplus \cdots \oplus q_{k-1}$        // $\oplus$ is bit-wise XOR
    return $q_1, \ldots, q_k$
Answer (query vector $q$, library $L$):
    // $q$ is one of the outputs of Query
    // $L$ has $n$ objects; each is $\ell$ bits
    // $q$ is a row vector, $L$ a logical matrix: $L \in \{0,1\}^{n \times \ell}$
    return $q \cdot L$      // product over the two-element field $\mathbb{F}_2$
Decode (answers $a_1, \ldots, a_k$):
    // $a_j$ is the output of Answer
    return $a_1 \oplus \cdots \oplus a_k$
```

Figure 2—The ITPIR protocol of CGKS [29]. n is the number of objects in library L, and ℓ is the length of each object. k is the total number of servers. (In Popcorn, k=2.)

- ***Communication efficiency.*** The length of the server's reply is $(\ell/y) \cdot |c|$ bits, where ℓ/y is the number of ciphertexts in the reply and $|c|$ is the size (in bits) of a ciphertext. $(\ell/y) \cdot |c|$ is comparable to ℓ, the size of object O_b, if the expansion ratio, $|c|/y$, of the underlying additively homomorphic cryptosystem is small.[2] The client's request contains n ciphertexts and is thus $|c| \cdot n$ bits. When $\ell \gg n$ (as will be the case in our context) and $|c|$ is a small constant (e.g., 2048 in many Paillier implementations), $|c| \cdot n$ is much smaller than ℓ.

2.4 Information-theoretic PIR (ITPIR) protocols

ITPIR protocols [29] use more than one server ($k > 1$), and assume that they do not collude; thus, in practice, the servers must belong to different administrative domains.

Figure 2 shows the CGKS [29] ITPIR protocol. It meets the three properties of PIR (§2.2):

- ***Correctness.*** The output of Decode is $\bigoplus_{j=1}^{k} a_j$, which equals $\bigoplus_{j=1}^{k} (q_j \cdot L)$. By properties of the field $\mathbb{F}_2$ (that addition is XOR and that multiplication distributes over addition), $\bigoplus_{j=1}^{k} (q_j \cdot L) = (\bigoplus_{j=1}^{k} q_j) \cdot L = e_b \cdot L = L[b]$.
- ***Privacy.*** Each server in $S_1, \ldots, S_{k-1}$ sees a randomly generated query vector, and therefore each server (and all of them combined) cannot learn any information about b. Server S_k sees q_k, which is constructed by XORing unit vector e_b with the one-time pad $q_1 \oplus \cdots \oplus q_{k-1}$. By the properties of one-time pads, S_k can learn information about e_b only by learning the one-time pad (or by colluding with all other servers).
- ***Communication efficiency.*** The combined length of the servers' reply is $k \cdot \ell$ bits. In Popcorn, we set $k = 2$ to keep this comparable to ℓ, the size of an object. A client's request consists of k n-bit-long query vectors, which is much smaller than ℓ when k is small.

[2] The Paillier cryptosystem has a message expansion ratio of ≥ 2.

	I/O	CPU	content prot. (ITPIR)	resists collusion	object sizes	pricing, reco
XPIR [12]		◐	●	●		
RAID-PIR [33]					◐	
Percy++ [49]		◐	◐	◐	◐	●
Popcorn	●	◐	●		◐	

Figure 3—Prior PIR-oriented works (rows) and which media-related challenges they address (columns), assuming two servers for ITPIR-based works. ● means that the work addresses the challenge; ◐ means that it partially addresses the challenge.

3 Challenges of applying PIR

Though PIR is promising, there are a number of challenges in applying it to large-scale media consumption:

- *Resources.* The I/O and CPU resources required to serve a single request are proportional to the size of the library. Batching requests should help amortize some of this overhead, but it is in tension with the next issue.
- *Strict deadlines.* Media delivery has stringent latency requirements: initial delay must be small, and the delivery must obey real-time constraints.
- *Variable object sizes.* Object sizes vary as a function of encoding or playback time. However, PIR assumes objects of identical size.
- *Content protection in ITPIR vs. CPIR.* Content creators may be loath to disseminate the content beyond its original distribution channel. Yet ITPIR requires multiple non-colluding servers, and hence multiple administrative domains, necessitating such dissemination. CPIR, on the other hand, requires only a single server; however, its computational cost is significantly higher.[3]
- *Billing, access control, recommendations.* For business reasons, media services may need to support access control, pricing policies (tiers, etc.), targeted advertising, and recommendations. Yet, private retrieval conflicts with all of this functionality.

Subsets of these challenges have been addressed before (Figure 3). Popcorn aims mainly at the resource consumption issue, via the architecture and design described next.

4 Architecture and design of Popcorn

Figure 4 depicts Popcorn's architecture. A *primary content distributor* creates an encrypted version of the library, L_{Enc}, using *per-object keys*, and replicates L_{Enc} to two *secondary content distributors*, each in separate administrative domains. The primary content distributor maintains a *key server*. Each secondary content distributor maintains an *object server* that is distributed over multiple physical machines.

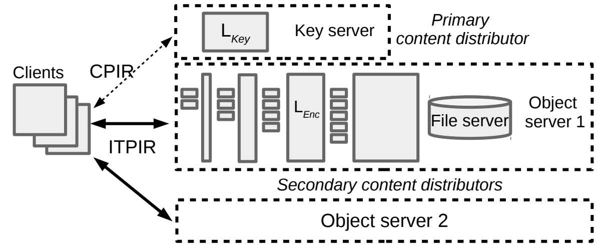


Figure 4—Popcorn's architecture. Popcorn uses two servers for ITPIR to keep ITPIR's network overhead (§2.4) small. Each object server stores all of the columns in the library (Figure 5), and is distributed over multiple physical machines.

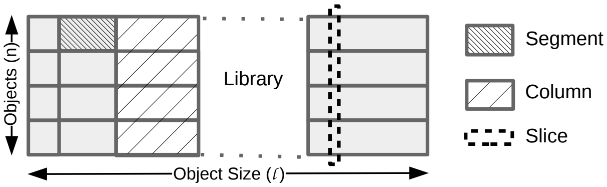


Figure 5—Popcorn terminology. Each column is stored by two ITPIR instances (one from each object server). Columns are divided into slices, which are assigned to physical machines.

The key server delivers the per-object keys using CPIR; the object servers deliver encrypted objects using ITPIR (§4.1). The distinction between key and object servers maps to today's DRM implementations [2, 3, 85], where clients contact two separate servers, one for encrypted video and one for decryption keys.

Media objects are split into *segments*—contiguous pieces of media containing, for example, a few seconds or minutes of a video. Segment sizes vary (§4.3). Each object is presumed to have the same decomposition into segments (we revisit this assumption in §4.4). The library is partitioned into *columns* (Figure 5); a column is a union of corresponding segments, across all objects. Therefore, a column's size is n times that of any segment it contains.

Each column is stored and served by two independent ITPIR *instances* (one for each object server); different instances use separate physical machines. Columns are further sub-divided into *slices*, which are the work units assigned to physical machines. A slice is 1 MB "wide" and n items "high"; we sometimes refer to 1 MB as a *chunk*. Each machine is responsible for one or more slices.

To retrieve an object, the client fetches a decryption key from the key server and the encrypted object from the object servers. The latter step proceeds in two overlapping phases. In the first phase, the client sends, in parallel, a query vector to all machines in both object servers. On receiving a request, a machine adds the query vector to a request queue. Each machine services its queue by: looping over its slices, computing chunk-sized ITPIR replies for every pending request, and pushing the resulting chunks to a file server (one per object server; Figure 4) that retains the chunks until they are requested by clients.

[3] The state of the art CPIR implementation is XPIR, which is based on the Ring-LWE cryptosystem. XPIR can process data at 22 Gbps on a machine with 4 physical (and 8 virtual) cores [12], while the CGKS ITPIR implementation in Percy++ [49], based on cheaper XOR operations, can process data at 152 Gbps on comparable hardware.

In the second phase, the client downloads these ITPIR-encoded chunks at the appropriate playback times, and applies Decode (Figure 2). This phase overlaps with the server-side generation of replies.

4.1 Composing ITPIR and CPIR

As stated earlier, Popcorn combines CPIR and ITPIR: the heavier-weight CPIR, which requires only one server, is used to serve per-object keys, while the lighter-weight ITPIR is used to serve the large encrypted objects. As a result, both keys and objects are served privately (because PIR is applied to them both), CPIR is not a performance bottleneck (because it is used only for small keys), and current controls on content protection are respected (because the plaintext content and keys are stored only at the primary content distributor).

As an alternative to CPIR, the key server could use Symmetric PIR (SPIR) or 1-out-of-n Oblivious Transfer (OT). Section 7 discusses these alternatives.

4.2 Batching

Popcorn uses the CGKS ITPIR scheme described in §2.4, as its inexpensive operations (XORs) keep its computational overhead low (by the standards of PIR). Still, because ITPIR queries are dense—on average, half of the entries are set to 1 (Figure 2)—responding to a query requires the machine serving a slice to read from storage and XOR, on average, $n/2$ chunks. This taxes I/O bandwidth, memory bandwidth, and CPU cycles.

To reduce costs, Popcorn's machines, which are oblivious to the content of queries, process queries in *batches*, and perform a single I/O pass over a slice for all of the queries in a batch. Batching thus amortizes I/O overhead and lets Popcorn exploit sequential transfer bandwidth.

Batching also reduces *computational* (not just I/O) overhead by leveraging the observation that the PIR computation required for a batch of requests can be expressed as matrix multiplication ($q \cdot L$ in Figure 2 can be replaced by $Q \cdot L$, where Q is a matrix whose rows are query vectors). Previous work [19, 71] (covered by the Percy++ row in Figure 3) has used this observation to incorporate sub-cubic algorithms [30, 58] that reduce the total number of operations required by PIR. Popcorn, by contrast, chooses block matrix multiplication [68], which, though it does not affect the total number of operations, leverages cache locality. One can view the resulting access pattern as batching at the CPU-memory interface.

4.3 Specializing batching for media delivery

Given the considerations in the previous subsection, Popcorn has an interest in increasing batch sizes (at least up to a point).[4] However, there is a tension between large batch sizes, which seem to require synchronizing clients, and meeting the deadlines of real-time media delivery. Popcorn resolves this tension as follows.

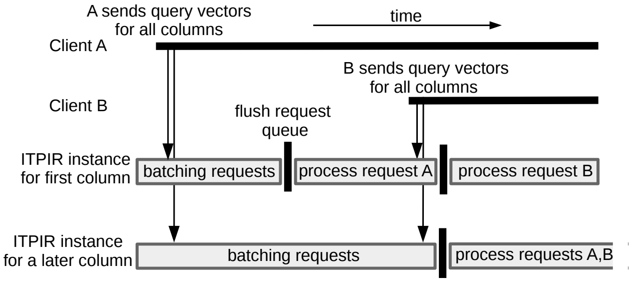

A sends query vectors for all columns
time
Client A
B sends query vectors for all columns
Client B
flush request queue
ITPIR instance for first column
batching requests
process request A
process request B
ITPIR instance for a later column
batching requests
process requests A,B

Figure 6—Batching at an object server in Popcorn. Requests to the initial column from two clients A,B are in separate batches as the processing cycle for this column is short. The requests to a later column (sent alongside the requests to the first) can be batched. This arrangement is inspired by Pyramid Broadcasting [102].

To begin with, each ITPIR instance loops over its assigned column (§4) continuously. Since a client can begin playback only after decoding the response for the first column, Popcorn uses a "narrow" first column to keep this initial delay short. Column width, however, increases quickly in Popcorn, making later columns wide. The crucial intuition is that *wide columns imply good batching opportunities*: a batch comprises all requests that reached an ITPIR instance during its previous loop interval, and wider columns imply longer loop intervals.

Figure 6 depicts this arrangement, which is inspired by Pyramid Broadcasting (PB) [102], wherein increasingly-sized pieces of a media object are served on separate broadcast channels. However, the details of our setup are different: the work of a Popcorn server depends on the number of clients (unlike in broadcasting), Popcorn relies on server-side buffering (PB relies on client-side buffering), and Popcorn is concerned with provisioning machines (PB concentrates on allocating bandwidth). These and other differences lead us to a solution that owes a debt to PB but is specific to our context.

Details. We start with two simplifying assumptions, which we revisit later: that a single ITPIR instance is handled by a single machine, and that there is no network delay or loss. Define an *instance processing cycle* as the duration of one iteration of an instance's loop. Within this cycle, an instance traverses each slice in turn, performing Answer for all queries that arrived during the prior cycle.

We want all clients to experience smooth playback. To this end, suppose that we are willing to impose startup delay d. Suppose further that $T_1 \leq d - \epsilon$, where T_1 denotes the processing cycle for the first instance, and ϵ is the time for the instance to handle a single slice. Likewise, define

[4] Above a certain batch size, there is no advantage: I/O is no longer a bottleneck, and the CPU benefits of using matrix multiplication stop increasing. However, there is also no disadvantage, so for simplicity, Popcorn does not bound batch sizes.

T_i as the processing cycle for the ith instance ($i > 1$), and suppose that for all such instances, $T_i \le d - \epsilon + \sum_{j=1}^{i-1} t_j$, where t_j is the playback time of segment j.

Under these conditions, we claim that any client, *regardless of when it joins*, experiences smooth playback. Why? Consider only instance 1: in the worst case, a client initiates consumption just after instance 1 begins its processing cycle. The client cannot download until the current processing cycle has terminated (which takes time T_1) and the first slice of the next cycle is processed (for an additional ϵ). Smooth playback simply requires the overall delay ($T_1 + \epsilon$) to be less than d, matching our conditions. Once playback begins, the client has t_1 addtional time before it needs the second segment. Generalizing, in the worst case for instance i (i.e., the client's initial request arrives just as a processing cycle begins), as long as T_i is no larger than $d - \epsilon + \sum_{j=1}^{i-1} t_j$ (which is exacly what our conditions guarantee), then the first slice of the ith instance will be ready, and playback will be smooth.

But how should the $\{t_i\}$ be set? Recall that, for more effective batching, Popcorn needs segment widths to increase: we are then seeking the maximum t_i for each instance i .

Let μ be the playback rate, P_i the rate at which XOR operations are processed by the ith instance, R_i the I/O bandwidth available to the instance, and b_i the batch size (the number of requests accumulated in a cycle of time T_i). To upper-bound t_i, we match load to capacity, for both I/O and CPU. For I/O, the column's data (n segments, each of size $t_i \cdot \mu$) is upper-bounded by the amount of data that the instance can read in one cycle: $t_i \cdot \mu \cdot n \le T_i \cdot R_i$. For CPU, the picture is similar, except that the total work scales with b_i, the number of clients being served: $t_i \cdot \mu \cdot n \cdot b_i \le T_i \cdot P_i$. These inequalities lead to:

$$t_i \le T_i \cdot \left(\frac{\min\{R_i,\, P_i/b_i\}}{\mu \cdot n} \right).$$

Assume that for all i, $\min\{R_i,\, P_i/b_i\} \ge \mu \cdot n$ (we will arrange for this in "Provisioning," below). Then, the foregoing bounds (on the $\{T_i\}$ and on load) imply that for all i, we can set:

$$t_i = T_i = 2^{i-1} \cdot (d - \epsilon)$$

(see Appendix A). Note that the $\{t_i\}$ increase exponentially in size, as desired. In particular, approximately half of the file is covered by the final segment.

Provisioning is driven by the earlier assumption that $\min\{R_i,\, P_i/b_i\} \ge \mu \cdot n$ for all i. To meet the requirements on R_i and P_i, Popcorn uses multiple machines per instance and aggregates their resources, by striping slices across them. If r_i is the per-machine I/O bandwidth for the machines used for the ith instance, then the I/O for instance i can be handled with $R_i/r_i = \mu \cdot n/r_i$ machines. P_i, the XOR processing throughput for instance i, increases with i because so does the batch size b_i; specifically, if λ is the overall rate at which clients initiate requests for objects, then $b_i = \lambda T_i$. Moreover, the per-machine XOR processing throughput for the ith instance, $p_i(\cdot)$, is a function of the batch size because cache locality in block matrix multiplication (§4.2) (and hence throughput) improves with a a bigger batch size. Thus, the task of processing the XOR operations for instance i can be handled by $P_i/p_i(b_i) = \mu \cdot n \cdot b_i/p_i(b_i)$ machines.

To account for the striping, we need to modify the earlier analysis of startup delay, smooth playback, etc.: if resources from k_i machines are aggregated for the ith instance, then each machine takes $\epsilon \cdot k_i$ time instead of ϵ to handle a slice. As a result, the inequality $T_i \le d - \epsilon + \sum_{j=1}^{i-1} t_j$ becomes $T_i \le d - \epsilon \cdot k_i + \sum_{j=1}^{i-1} t_j$, and both the $\{T_i\}$ and $\{t_i\}$ are computed accordingly.[5]

The total number of machines, across all I instances, is: $\mu \cdot n \cdot \sum_{i=1}^{I} \max\{1/r_i,\, \lambda T_i/p_i(\lambda T_i)\}$. Notice that if the max is controlled by the first term, then the given instance is bottlenecked by I/O (and the CPU resource is sometimes idle); if by the second, then the instance is bottlenecked by CPU work (and the I/O resource is sometimes idle). Later (§6.1) we will obtain estimates empirically for r_i and $p_i(\cdot)$.

Popcorn must also provision for the file server machines (§4). The file server requires the buffer space for each instance to equal the number of requests in service times the size of a segment, i.e., $\sum_{i=1}^{I} b_i \cdot (t_i \cdot \mu)$. The file server also requires I/O bandwidth equal to the rate at which reply data is produced and consumed: $2 \cdot \sum_{i=1}^{I} b_i \cdot \mu$ (assuming $t_i = T_i$).

Finally, we have been assuming no burstiness or delay in the network. To account for network fluctuation, we must allow for clients to build up a playback buffer, of some time length β. To this end, T_i should be upper-bounded by $d - \epsilon \cdot k_i - \beta + \sum_{j=1}^{i-1} t_j$, and the $\{t_i\}$ computed to be consistent with T_i.

Discussion. To understand the savings and amortization from Popcorn's batching, consider a naive batching scheme, in which time is divided into *epochs* of length T_{epoch}. Let a *cohort* denote the set of clients who initiate a request (for the first chunk of a media file) in an epoch. Then, the entire cohort moves through the slices, as it were, together. Each cohort needs enough machines to meet two requirements: (a) $\mu \cdot n$ I/O bandwidth, and (b) $\mu \cdot n \cdot \lambda \cdot T_{\text{epoch}}$ XOR processing throughput (here $\lambda \cdot T_{\text{epoch}}$ is the cohort's batch size). If $H = T/T_{\text{epoch}}$ is the total number of cohorts (where T is the total playback time), then the total number of machines is

[5] The computation must resolve a circular dependency as T_i is expressed in terms of k_i, which itself depends on the segment size, with a bigger segment requiring more machines. We resolve this circularity by repeating the process of speculatively setting a k_i, calculating T_i, and then refining the speculated value of k_i using the obtained value of T_i.

$\mu \cdot n \cdot \sum_{i=1}^{H} \max\{1/r,\ \lambda \cdot T_{\text{epoch}}/p(\lambda \cdot T_{\text{epoch}})\}$, where r is the per-machine I/O bandwidth, and $p(\lambda \cdot T_{\text{epoch}})$ is per-machine XOR processing throughput for a batch size of $\lambda \cdot T_{\text{epoch}}$. Here, T_{epoch} must be upper-bounded by $d - \epsilon \cdot k - \beta$ to meet the startup delay requirements, where k is the number of machines for a cohort.

To compare the cohort batching scheme to Popcorn, we make the simplifying and optimistic assumption that both schemes use machines that make the two terms of the max equal, so that no resources are idle (we will revisit this assumption in §6.2 and §6.4). Then, the total I/O bandwidth required by the cohort scheme is $H \cdot \mu \cdot n$, which is considerably larger than what Popcorn needs ($I \cdot \mu \cdot n$, where $I \ll H$).

In terms of computational resources, the cohort scheme needs $\mu \cdot n \cdot \lambda \cdot T/p(\lambda \cdot T_{\text{epoch}}) = \mu \cdot n \cdot \lambda \cdot \sum_{i=1}^{I} T_i/p(\lambda \cdot T_{\text{epoch}})$ machines; Popcorn requires instead $\mu \cdot n \cdot \lambda \cdot \sum_{i=1}^{I} T_i/p_i(\lambda \cdot T_i)$ machines. Neither scheme is the clear-cut winner; however, if we assume that $p(\cdot) = p_i(\cdot)$ for all i, then Popcorn has lower computational demands, because (a) $T_i \approx 2^{i-1} \cdot T_{\text{epoch}}$ (by our earlier analysis) and (b) $p(\cdot)$ is monotonically increasing. In essence, Popcorn has larger batches, so (holding machine type configuration constant) the benefit of locality is more pronounced (§4.2), lowering Popcorn's computational requirements relative to the naive batching scheme.

4.4 Handling variable-sized objects

The design has so far assumed equally sized objects. Unfortunately, the naive solution—padding all objects to the size of the longest one—is inefficient for Netflix: it causes a 4× increase in network transfers (the average movie is approximately 1.5 hours while the longest is 6). Moreover, to not reveal the true object size, a client must download the padded content in full.

Popcorn's solution combines padding and compression. It chooses a representative object O_{avg} from the library (for example, the object closest to the average media length) and makes all other objects O_{avg} in size: objects smaller than O_{avg} are padded. Longer objects, meanwhile, are compressed by reducing the bitrate.

Both padding and compression are potentially problematic. Padding could waste resources, but by using it in combination with compression, Popcorn ensures that objects will be padded only up to O_{avg}, limiting costs. For compression, we consider two cases. First, if the object is up to 30% bigger than O_{avg}, then the compression required to make it O_{avg} is small, and the consequent degradation in quality is likely to be tolerable. Indeed, several studies [41, 64, 94] suggest that small variations in video bitrate have a limited impact on user satisfaction.

The second case is that the object is much larger than O_{avg}; a trade-off then arises between quality and on-demand consumption. On the one hand, Popcorn can compress these objects aggressively; however, doing so will result in significant quality degradation. On the other hand, Popcorn, like others [33, 54], can divide up these objects; however, the client would then have to download each division as if it were a separate movie, which means delaying consumption or downloading far ahead of time (if the separate divisions were downloaded all at once, then an attacker could guess that a longer object is being consumed).

The Netflix catalog [1] indicates that the majority of movies have a similar size: 85% of the objects are between 60 and 120 minutes, with the majority clustered around the average movie length of 92 minutes. Movies between 92 and 120 minutes will consequently require, on average, 10% compression. Similarly, the padding for objects between 60 and 92 minutes will be small. The impact of objects at either extreme will be limited: 8% of the movies are below 60 minutes, and will require significant padding; 6% are between 120 and 150 minutes, making them candidates for aggressive compression (29% on average); 1% are over 150 minutes, making them candidates for splitting. We think that splitting is not a huge limitation, because we hypothesize that people usually plan ahead to watch long movies.

5 Implementation

Our prototype implements the design in Section 4, except for large file splitting (§4.4). It leverages existing PIR implementations: the key server uses the XPIR [12] implementation of the CPIR protocol in Figure 1. For the object servers, we borrow the CGKS ITPIR implementation of Percy++ [49][6] and modify it to support the techniques in Section 4. The total server-side code is 11K lines of C++. We implement two versions of the client-side library: one in C++ (2500 lines), which we use for experiments (§6.2), and one in JavaScript (500 lines), which we use to show compatibility with modern web browsers (§6.5).

6 Evaluation

Our evaluation answers the following questions:

1. When is Popcorn affordable?
2. What is the price of Popcorn's privacy guarantees?
3. Can we use Popcorn to watch a movie encoded using an existing DRM scheme on a modern web browser?

Figure 7 summarizes our evaluation results.

Method and setup. We compare Popcorn to three baselines. *NoPriv*, *BaselinePIR*, and *BaselinePIR++*. NoPriv serves object chunks from an Apache web server, modeling modern media delivery systems that use HTTP

[6] Percy++'s CGKS ITPIR implementation is one of the fastest implementations for two-server ITPIR. An alternative is the CGKS implementation from RAID-PIR [33] (§7).

Popcorn is affordable when it serves large media files to many concurrent clients.	§6.2
Popcorn's per-request dollar cost is 3.87× of a system without privacy for workloads with $\geq$10K concurrent clients.	§6.3
Popcorn integrates well with existing web technology. It can play DRM-encoded media within modern web browsers.	§6.5

Figure 7—Summary of main evaluation results.

	type	vCPUs	RAM (GB)	SSDs (# × GB)	cost/hr
c3.8xl	1	32	60	2 × 320	\$0.6281
i2.4xl	2	16	122	4 × 800	\$0.8451
i2.8xl	3	32	244	8 × 800	\$1.6902

Figure 8—Hourly cost of reserved Amazon EC2 machines used in our experiments. Machines starting with "c" are compute-optimized; those starting with "i" are I/O-optimized.

caching at CDN edge servers [11]. BaselinePIR is a modified version of Percy++ [49] CGKS: the servers store the library L as slices and process ITPIR queries directed at them. This is essentially Popcorn without the techniques of §4. BaselinePIR++ additionally batches requests using cohort batching (§4.3) to reduce both I/O and CPU costs. For all PIR systems, we experiment with one object server and multiply the measurements by two (to reduce the financial cost of our experimental evaluation).

Our workload is modeled on existing media delivery services [99]: clients arrive according to a Poisson process (e.g., C=10K clients arrive in T=90 minutes). All clients in NoPriv request the same (average-size) object, giving this baseline the maximum benefit of server-side caching. The server's work in Popcorn, BaselinePIR, and BaselinePIR++ is oblivious to the request distribution (we select a Zipfian distribution with θ=0.8).

For the four systems, we measure server- and client-side resource usage in terms of CPU time (by instrumenting code with `clock()`), I/O transfers and storage (using `iostat`), and network transfers (via `/proc/net/dev`).

Our experimental testbed is a single availability zone within Amazon's EC2, and is described in Figure 8.

6.1 Provisioning resources using microbenchmarks

Popcorn. Machine provisioning for Popcorn involves two steps: (1) benchmarking the basic operations (see Figure 9 caption for details), and (2) combining the results with the provisioning analysis in §4.3.

Consider, for example, provisioning the first ITPIR instance of a Popcorn object server for a Netflix-like workload: C=10,000 clients streaming from a library of n=8192 media files with average playing time of T=90 minutes, playback rate of μ=4 Mbps, and startup delay of d=15 seconds.[7] The processing cycle of this instance must be $T_1 \leq d - \epsilon \cdot k_1$ (§4.3). For our example, ϵ=2 (the time to process or consume a 1 MB chunk at μ=4 Mbps), and we speculatively set k_1=3, which gives $T_1 \leq 15 - 2 \cdot 3 = 9$ seconds. Thus, the instance is given a segment of t_1=T_1=9 seconds and has a batch size of $b_1 = (C/T) \cdot T_1 = 17$. Furthermore, it requires storage capacity of $n \cdot t_1 \cdot \mu$=36 GB, read bandwidth R_1=$n \cdot \mu$=32 Gbps, and XOR processing throughput $P_1 = b_1 \cdot n \cdot \mu = 544$ Gbps.

	Throughput (Gbps)		
	c3.8xl	i2.4xl	i2.8xl
Sequential read	6.4	12.6	23.3
Random mixed rw	2.1	8.0	16.0
block matrix multiplication	488–4968	488–2512	432–4608

Figure 9—Throughput of basic operations in Popcorn—reading a column slice (§4.3), reading and writing 1 MB sized chunks, and computing block matrix multiplication on a slice (§4.2)—on machines listed in Figure 8. The latter value depends on the size of the query matrix (§4.2, §4.3), so we report a range—from a query matrix consisting of a single query vector to one that contains 4096 query vectors.

Our microbenchmarks (Figure 9) indicate that these requirements can be met by three i2.4xl machines. Had we required a different number of machines than three, then, as described in §4.3, we would have had to adjust k_1 and repeat the provisioning process described above.

BaselinePIR. To use the fewest possible machines, we stripe the approximately 21 TB library of our Netflix-like workload across four i2.8xl instances.

To reduce the financial cost of our experimental evaluation, we measure the number of requests that can be serviced by this setup, along with each request's resource consumption, and extrapolate the results to workloads with a larger number of requests (e.g., to support 2× concurrent clients, we double resource costs).

BaselinePIR++. As in Popcorn, we use the microbenchmarks in Figure 9 and the provisioning analysis for the cohort batching scheme (§4.3).

6.2 Per-request overheads of Popcorn

To understand when Popcorn is affordable, we run experiments varying the number of concurrent requests (C); the number of objects (n); and the playing time of objects (T). We find that Popcorn incurs modest costs when the library size is moderate ($\approx$8K media files), object sizes are large ($\approx$90 minutes), and there are many concurrent clients ($\geq$10,000). Fortunately, these settings are consistent with the workloads of Netflix-like systems (§8).

Before proceeding, we note that Popcorn's provisioning method can leave resources idle (§4.3), so we report both the consumed and provisioned resources. We focus on the consumed resources in this subsection and account for the idle resources in the next subsection.

[7] We think that 15 seconds of delay before playing a long video is tolerable. During this time the server could display a generic advertisement or public service announcement (existing services commonly display 15 or 30 second advertisements [10]).

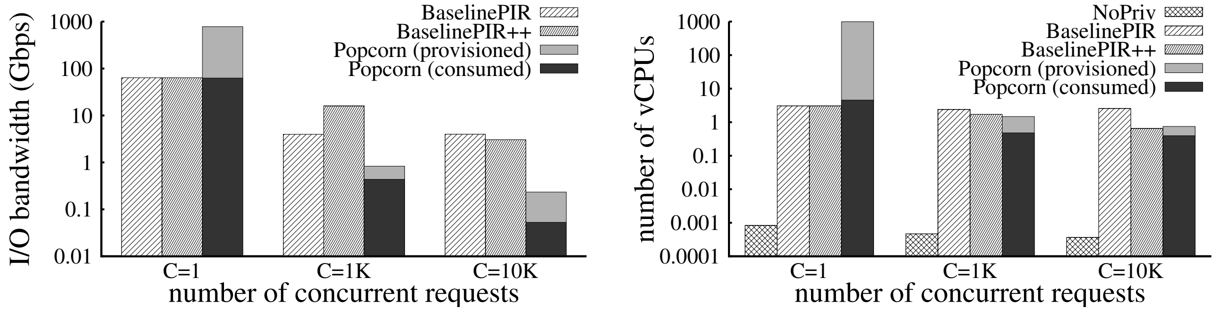


Figure 10—Per-request server-side resource use (log-scaled) of Popcorn and the baselines with varying concurrent requests C. If I/O is the bottleneck, there are idle CPU cycles and vice versa (§4.3). For Popcorn, we depict both the provisioned and consumed resources; for the baselines, we depict only the latter. We do not depict I/O usage for NoPriv as it is always zero (see text).

Overhead versus number of concurrent requests. We run Popcorn and its baselines with C=$\{1,1\text{K},10\text{K}\}$ while keeping n=8192, T=90 min, μ=4 Mbps, and d=15 seconds. Figure 10 summarizes the per-request server-side resource costs.

I/O overheads. When C=1, the I/O bandwidth Popcorn consumes matches that of BaselinePIR and BaselinePIR++, as there is no opportunity to batch requests. However, as the request rate increases, batching lets Popcorn amortize its I/O transfers (§4.3): the per-request amortized I/O bandwidth decreases from ≈ 63 Gbps (for C=1) to 53 Mbps (for C=10K), a reduction of 1190×. Surprisingly, concurrent requests, by hitting the file system cache, also reduce BaselinePIR's per-request I/O bandwidth (by 16×). As expected, BaselinePIR++'s per-request I/O bandwidth reduces by the cohort batch size. Finally, there are no I/O transfers in NoPriv as all requests hit the same (cached) object.

CPU overheads. For a single request, Popcorn consumes 50% more CPU than BaselinePIR, as the overhead of parallelizing block matrix multiplication (over multiple cores) in Popcorn (§4.2) is charged to a single request. As the number of concurrent requests increases, Popcorn's CPU overheads decrease; the per-request CPU consumption decreases by ≈11× when the number of concurrent requests increases from 1 to 10,000. We hypothesize that this stems from the increase in cache locality from block matrix multiplication over bigger batch sizes.[8] Furthermore, the 36 minutes of per-request CPU time for C=10K matches the performance of the matrix multiplication microbenchmark (42 TB of data processed in 36 minutes gives a throughput of 159 Gbps for a single CPU, consistent with the throughputs reported in Figure 9).

However, Popcorn's per-request CPU consumption is much higher than NoPriv (1080× for C=10K): for a single object, the Apache web server in NoPriv serves 1 MB chunks and requires almost no server-side processing, whereas Popcorn XORs n objects on average.

Network and storage overheads (not depicted in the figures). BaselinePIR, BaselinePIR++, and Popcorn incur a two-fold network overhead over NoPriv because clients download from two servers. With respect to storage, each instance of an object server in Popcorn needs buffer space equal to its segment size times its batch size (§4.3). Across all instances, this equals ≈15.4 TB, or ≈1.6 GB per concurrent request, which is 0.6× the size of an object.

Overhead versus number of objects. In Figure 11(a) we change the size of the library (n=$\{2048,4096,8192\}$) while keeping the other parameters fixed (C=10K, T=90 min, μ=1 Mbps,[9] and d=15 seconds). As expected, Popcorn's per-request CPU and I/O bandwidth consumption, even though amortized, is proportional to n. Network downloads and server-side storage overheads (not shown) do not change with n.

Overhead versus playing time of objects. In Figure 11(b), we change the playing time of objects (T=$\{10,60,90\}$ minutes) while keeping the other parameters fixed (n=2048, μ=1 Mbps, d=15 seconds, and C=10K). As T increases, the number of CPUs consumed per-request is unaffected, while the consumed I/O bandwidth decreases. Thus, for bigger objects, Popcorn's efficiency (in terms of consumed I/O bandwidth) improves. However, we note that idle I/O bandwidth (not depicted in the figure) also increases for bigger objects (§4.3).

Overheads of the key server. Recall that Popcorn uses XPIR [12] as its CPIR implementation (§5). Since XPIR does not batch requests, the per-request overheads of the key server depend only on the number of keys (and not on the number of concurrent requests C). We use a single machine of type c3.8xl for the key server. For a library with 8192 keys, it takes three seconds of server-side CPU time to privately retrieve a key; there are no I/O transfers

[8] In a separate experiment, we measured the percentage of cache misses for block matrix multiplication (§4.2) using CPU performance counters, and found that it reduces from 48% for a query matrix with a single request to less than 2% for a query matrix with 2^{10} requests.

[9] To reduce the financial cost of EC2 experiments, this and subsequent experiments set μ=1 Mbps instead of 4 Mbps. The change scales down the experiments; the qualitative results are unaffected.

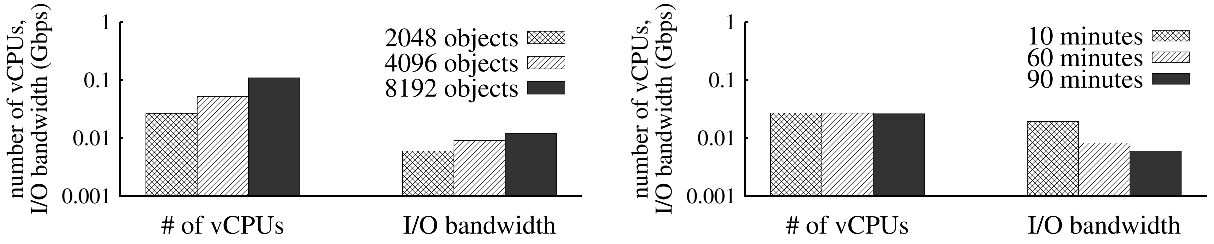


Figure 11—Popcorn per-request resource use (log-scaled) as a function of the number (left) and length (right) of objects.

	experimental configuration				per-request costs ($)		
	#reqs	#1	#2	#3	machine	network	total
NoPriv	10K	–	–	–	–	0.016	0.016
Popcorn	1	2	60	0	77.943	0.032	77.975
Popcorn	1K	17	50	4	0.09	0.032	0.122
Popcorn	10K	185	32	32	0.03	0.032	0.062

Figure 12—Estimated per-request dollar cost for NoPriv and Popcorn. #1, #2, and #3 refer to the type of AWS EC2 machines from Figure 8.

because the 128 KB library fits in memory. Thus, as expected, the key server is not a performance bottleneck for Popcorn. Moreover, the end-to-end time to retrieve a key is much less than the startup delay of $d=15$ seconds.

Client-side overheads. Compared to NoPriv, Popcorn's client consumes additional CPU and network bandwidth (because it has to generate and decode PIR queries, and download content from two object servers). For $n=8192$ objects, $T=90$ minutes, and $\mu=4$ Mbps, we find that Popcorn's client (run on a single VCPU of c3.8xl type machine) consumes 10 CPU seconds (compared to NoPriv's 1.7 CPU seconds), and 25 MB of network upload bandwidth (compared to NoPriv's 11 MB).

6.3 Dollar-cost analysis

The previous subsection showed that Popcorn significantly reduces CPU and I/O consumption over the baseline PIR systems, at least for large objects and high load. These improvements provide the foundation for achieving privacy at low cost, a cost that we now quantify.

Method. We use the pricing model of Amazon EC2 (Figure 8) to estimate the per-request machine cost, and the pricing model of CDNs ($0.006 per GB) [87] to compute per-request network cost. We choose these pricing models because they are public—though, in an actual deployment, a service could receive wholesale, bulk, or negotiated prices. We use a Netflix-like workload in our calculations: $n=8192$ media files, $T=90$ minutes, $\mu=4$ Mbps with varying number of concurrent clients. Figure 12 summarizes our results. We find that Popcorn's per-request cost is within a small multiple of NoPriv for a workload with $C=10$K concurrent clients.

NoPriv. To give NoPriv the maximum benefit, we disregard its machine cost. The per-request cost is then determined solely by the network transfer cost, and is ≈$0.016 (i.e., 90 minutes × 4 Mbps × $0.006/GB).

Popcorn. We provision EC2 machines as described in §6.1 and §4.3. The total per-request cost is derived by combining (1) the per-request machine cost, computed by dividing total machine cost by the total number of requests, and (2) the per-request network cost. This method charges Popcorn for both consumed and idle resources (Figure 10). For the Netflix-like library and $C=10$K, the per-request cost is $0.062 (the per-request machine cost is $0.03; the per-request network cost is $0.032).[10] Popcorn thus increases dollar cost 3.87× over NoPriv, in line with our initial affordability requirement (§1). Importantly, Popcorn's low cost is premised on many clients accessing the system concurrently: the per-request machine cost decreases with the number of concurrent clients. It is $78 for $C=1$ and $0.09 for $C=1$K.

BaselinePIR and BaselinePIR++. Since we might have provisioned these systems wastefully, we do not estimate their dollar cost using the machine-based pricing model, which charges for both the consumed and idle resources. Instead, we use a per-resource pricing model to estimate the dollar cost of these systems, as described next.

6.4 Further comparisons

In this subsection, we estimate the dollar cost of BaselinePIR, BaselinePIR++, XPIR [12], and XPIR++, a hypothetical extension to XPIR that uses cohort batching (§4.2) to reduce I/O costs (but does not use matrix muliplication). Figure 13 summarizes these alternatives.

The estimates for BaselinePIR, BaselinePIR++, and Popcorn are based on experimental data from §6.2; for XPIR and XPIR++, we calculate CPU resource consumption using XPIR's reported performance and I/O bandwidth consumption from the expression $2\cdot(n\cdot\mu)/b_{\text{cohort}}$ (the factor of two is due to XPIR's preprocessed library being twice the size of the original [12]).

We compare these systems for the Netflix-like workload of §6.3. We set the startup delay d to 15 seconds, except for the systems using cohort batching scheme, for which we vary d.

[10] The network cost can be reduced for a pricing model in which network transfers between (ITPIR) servers is cheaper than server to client transfers, by using techniques of Riffle [67, Section 4.4].

system	description
XPIR [12]	fastest CPIR implementation
XPIR++	XPIR with naive batching (§4.2)
BaselinePIR	XPIR composed with CGKS ITPIR (§4.1)
BaselinePIR++	BaselinePIR with naive batching (§4.2)
Popcorn	§4.1+§4.2+§4.3

Figure 13—Comparison points. "Naive batching" refers to an instantiation of batching, as described in §4.2, with the cohort batching scheme described in §4.3.

	# vCPUs	I/O bandwidth (Gbps)	Dollar cost relative to NoPriv
X [12]/X++ ($C=1$)	11.6	64	265×
X++ ($C=1$K)	11.6	26.6	118×
X++ ($C=1$K, $d=60$)	11.6	5.96	37×
X++ ($C=1$K, $d=600$)	11.6	0.58	16×
X++ ($C=10$K)	11.6	2.66	24×
X++ ($C=10$K, $d=60$)	11.6	0.59	16×
X++ ($C=10$K, $d=600$)	11.6	0.058	13.5×
B/B++ ($C=1$)	3.1	64	256×
B ($C=1$K)	2.4	4	19×
B ($C=10$K)	2.5	4	19×
B++ ($C=1$K)	1.7	16	66×
B++ ($C=1$K, $d=60$)	1.26	9.15	39×
B++ ($C=1$K, $d=600$)	0.49	0.54	4.5×
B++ ($C=10$K)	0.65	3	14×
B++ ($C=10$K, $d=60$)	0.49	0.59	4.7×
B++ ($C=10$K, $d=600$)	0.41	0.058	2.5×
P ($C=1$)	4.6–992	63–781	253×–4873×
P ($C=1$K)	0.5–1.47	0.43–0.83	4×–7.6×
P ($C=10$K)	0.4–0.74	0.053–0.23	2.5×–3.87×

Figure 14—Per-request resource consumption and estimated dollar-cost of XPIR (X), XPIR++ (X++), BaselinePIR (B), BaselinePIR++ (B++), and Popcorn (P). Network transfers are not shown; they are 5× NoPriv for X and X++, and 2× NoPriv for the other systems. For Popcorn, we present a range: the smallest value considers only the consumed resources, while largest value includes both consumed and idle resources. Startup delay d is 15 seconds unless specified otherwise.

We use a per-resource pricing model (derived in Appendix B) based on Amazon EC2's machine cost (Figure 8) and on the network cost of CDNs [87]. Our model charges CPU at $0.0076/hour, I/O bandwidth at $0.042/Gbps-hour, and network transfers at $0.006 per GB. Multiplied by each system's consumption of the corresponding resources, these values determine the per-request dollar cost (Figure 14).

- The costs of XPIR are high (265× NoPriv), though adding a naive batching scheme (XPIR++) significantly reduces them (by ≈11× for $C=10$K, $d=15$).
- Using ITPIR for object delivery (in conjunction with CPIR (§4.1)) reduces the costs further (by ≈ 2× for $C=10$K, $d=15$). The disadvantage is that ITPIR requires non-colluding servers.
- Increasing the startup delay (and thus the batch size of the cohort) can further reduce costs. For example, increasing d from 15 to 60 seconds reduces costs by 3× (a reduction from 14× NoPriv to 4.7× NoPriv).
- BaselinePIR++ matches the cost of Popcorn (when $C=10$K) but requires a 40× higher startup delay ($d=10$ minutes in BaselinePIR++ vs. 15 seconds for Popcorn).

6.5 Compatibility study of Popcorn

To verify Popcorn's compatibility with modern Web browsers and DRM technology, we implemented a Popcorn client in JavaScript and used it to watch short videos in the WebM format [8] (protected using WebM Encryption [9]). Our prototype works on Chrome (version 45.0.2454), and makes use of the HTML5 video tag and extensions: the decoded ITPIR content is passed into the Media Source Extension interface, which forwards media chunks to the video player; the decoded CPIR response is passed into the Encrypted Media Extension interface, which decrypts the protected content.

7 Related Work

Alternatives to PIR for privacy. *Obfuscation* [18, 42, 88] protects clients' privacy by cloaking traffic with dummy requests. This approach requires less processing than PIR at clients and servers, but significantly higher network cost: matching PIR's degree of privacy (the number of objects among which a request is hidden) would require downloading the entire library.

Rather than the content being consumed, *anonymity* hides the identity of the consumer [39, 69]. This could be used to hide metadata (login times, download frequency, etc.), which is complementary to PIR. However, anonymity-based solutions can reveal access patterns that, combined with other background information, may disclose a user's media consumption [77].

Oblivious RAM (ORAM) [51, 72, 74, 97] algorithms conceal a client's access patterns from a storage server. Similarly, *searchable symmetric encryption* (SSE) (surveyed in [26, 27]) offers yet another solution for private data retrieval from a remote database. However, these solutions target a setup where the client outsources its encrypted data to a server.

Recent results [61, 84] enhance the above setup: they let clients privately retrieve data from a remote database owned by a *different* entity. Unlike PIR, these protocols allow for a controlled amount of leakage in the form of data-access and query patterns. Unlike us, they assume that the server does not collude with clients (e.g., in Popcorn the server can pretend to be a new customer of the streaming service). If the server can collude with a client, it can issue queries for each media file in the system, monitor access patterns, and decode all other clients' queries.

Improving the performance of PIR. The computational challenges of PIR have been obvious since its introduction, and have since been mitigated in several ways.

Distributing the work, either by moving it to the cloud or by dividing it among clients [35, 73, 83], reduces latency but not the total computational burden.

GPUs [31, 75] and cheaper cryptographic operations [12, 13, 43, 101, 107] have reduced the computational load of CPIR, refuting the notion [95] that CPIR is likely to be more expensive than the naive solution of transferring the entire library. However, the single request cost for media delivery in XPIR [12], the fastest system employing these techniques, is still higher than desirable (see §6.4 and Figure 14 for a comparison with Popcorn).

Another path to better performance is to limit the privacy guarantees to only a portion of the library [79, 80, 103]. For example, bbPIR [103] allows users of libraries that can be thought of as a matrix to specify a submatrix (called a bounding box) from which bits can be privately retrieved using CPIR. This approach can be useful for efficiently implementing privacy-preserving location-based services: the larger the bounding box, the higher the privacy, but also the higher the processing and network costs.

Perhaps the most direct way to reduce the overhead of PIR is to genuinely reduce the work that servers need to perform. Lueks and Goldberg [71], building on earlier theoretical work by Beimel et al. [19] and Ishai et al. [60], show that one can achieve sub-linear server-side computation by efficiently processing batches of requests from multiple clients. Popcorn is inspired by this work: it uses batching at multiple stages of its protocol, but tailored for media delivery. Another recent system, RAID-PIR [33], based on the implementation of upPIR [25], reduces server-side work, first, by storing and processing only a *fraction* of the library at each ITPIR server and, second, by encoding multiple requests from the same client in a single query. Popcorn's performance could potentially benefit from these techniques, but only when using more than two servers, or when clients issue multiple simultaneous requests. Currently, Popcorn assumes exactly two servers and that clients request objects sequentially.

Finally, performance can be improved with dedicated hardware [16, 59, 70, 96, 105], at the price of having to trust its manufacturer: a client can connect to a secure coprocessor that (obliviously to the server hosting the library) retrieves and delivers the requested object.

A large body of literature focuses on instead reducing the communication overhead of PIR [45, 81]. Unlike Popcorn, these protocols target an environment in which $n \gg \ell$. In that context, Devet et al. [36] propose a technique that, like Popcorn, composes CPIR and ITPIR. Unlike Popcorn, the composition is hierarchical (ITPIR selects a sub-library, and iterations of CPIR select an object) and minimizes communication costs.

Protecting library content in PIR. The tension between ITPIR and content protection has been noted before. Gertner et al. [47] introduce the problem and propose two solutions, both of which, at a high level, protect the content by storing at untrusted servers independent random data (e.g., two servers store random data that XORs to the library content). Goldberg's ITPIR protocol [50] has a similar protection property as [47], but it uses fewer servers. Huang et al. [57] protect library content kept at untrusted servers by first encrypting it, and then using a threshold signature scheme [34] to serve keys for the encrypted object. In all the above schemes, the library content can be disclosed if more than a threshold of untrusted servers collude. By composing CPIR and ITPIR (§4.1), Popcorn instead keeps content protection collusion-proof.

Symmetric PIR (SPIR) schemes add an additional facet to content protection by preventing dishonest clients from learning information about the content of a database beyond what is contained in the records they retrieved [48]. Popcorn currently assumes an honest client (§2.1) and thus does not use SPIR to privately download keys from the key server; however, it can reduce that trust by transforming its CPIR protocol into an SPIR protocol [38, 76].

1-out-of-*N* oblivious transfer (OT) [21, 76] provides the same content protection property as SPIR but, unlike SPIR, can have network overhead linear in the size of the library. In our experiements, this overhead would not be costly: WebM Encryption (§6.5) sets our keys to 128 bits, which, for n=8192 objects, yields a library of only 128 KB. However, the linear overhead can in general be large (e.g., if the key server embeds keys within DRM licenses; for this reason, Popcorn's key server does not use OT.

Handling variable-sized objects in PIR. A naive solution is to pad every object to the size of the longest, and download (the equivalent of) the longest object from each server. Prior work [33, 54] avoids this solution by (a) concatenating small objects (e.g., a few objects form one row of the library), and (b) splitting large objects over multiple rows of the library and using *multi-row queries* that retrieve (secretly) many rows in a single query. The reduced communication cost is close to the optimal: the size of the longest object in the library. However, this cost is still high, especially if a smaller object is being retrieved. An alternative is to download different rows (of an object) as independent objects, possibly at the cost of increasing the consumption delay [33]. Popcorn uses this technique for objects that are divided over multiple rows, but in addition reduces the number of such objects by using a combination of compression and padding (§4.4).

Prior PIR implementations. Many of the CPIR and ITPIR protocols described above have been implemented. The Percy++ library [49] contains several of them [13, 29, 35–37, 50, 55, 71]. Also, [54] is implemented as a fork of Percy++, RAID-PIR [33] is implemented on top of upPIR [25], and there are numerous CPIR implementations [12, 31, 43, 73, 75, 83, 90, 101, 103, 107], among which XPIR [12] is the fastest. Popcorn incorporates some of these implementations as modules: it uses the XPIR library for CPIR and borrows the CGKS ITPIR [29] code from Percy++. Sections 5 and 6 empirically or analytically compare Popcorn against these prior implementations.

8 Discussion, limitations, and future work

We evaluated Popcorn at the scale of a Netflix library, and found that the results are cautiously encouraging: compared to a baseline, I/O and CPU overhead are both lower (due to amortization, batching, and careful provisioning). And, although the overall resource cost is high, the *dollar* cost is manageable. Below, we discuss fundamental limitations of Popcorn, followed by limitations of the prototype and current design that require future work.

Fundamental limitations. We see three main limitations. First, because Popcorn's overheads grow linearly with the number of objects, it has no hope of scaling to YouTube-size libraries. Second, organizations that serve objects can collude to compromise Popcorn' privacy guarantee. Admittedly, the no collusion assumption on which rests Popcorn' privacy guarantee may be unrealistic against state-sponsored adversaries that can compromise multiple organizations (or already have). Third, Popcorn cannot support forward seeks during playback: such user actions alter the download pattern in a content-dependent way, thus revealing information.

Library updates. To support online updates, Popcorn should execute both CPIR and ITPIR queries on the same version of the key and object libraries, at the key server and at both object servers. Standard solutions exist (e.g., generation numbers in concert with garbage collection).

Integration with CDNs. Running Popcorn on content delivery networks (CDNs) would present two main challenges: maintaining the utility of batching when running on a distributed infrastructure; and increased hardware provisioning at the CDN's edge servers (power consumption, colocation space, etc.). Though addressing the latter is non-trivial, we believe that it does not require a paradigm shift: Akamai's EdgeComputing service [32] already enables running CPU-intensive enterprise business web applications at edge servers. Moreover, Netflix recently installed custom-built storage-optimized appliances at the edges.

Similarly, we believe that, though the CDN's distributed infrastructure will reduce opportunities for batching, enough concurrency will remain to make the service cost effective. Indeed, rough back-of-the-envelope calculations suggest that request rates for Netflix are already quite high (e.g., over 9200 requests/90min/PoP[11]) and are growing fast [5]. This is not specific to Netflix: similar request rates (average of 6000 requests in 90 minutes from within a single city) have been reported for other video on demand systems [108].

Changes in load. Unless Popcorn is always wastefully provisioned for the peak load, load changes require care: the assignment of work units to machines depends on the number of clients (§4.3). A solution is to rely on virtual machines (VMs): give each VM a single slice, and then provide elasticity via VM migration or consolidation.

Variations in quality and bandwidth. *Adaptive streaming* lets clients dynamically switch between different video quality levels to adjust to bandwidth fluctuations. Popcorn could support this feature in two ways. First, it could maintain an individual library for each video quality level (where each library is adapted for Popcorn as described in §4.4). Clients would send query vectors to all libraries but download a given video chunk only from the appropriate one. Switching between libraries doesn't leak information if download pattern is fixed (e.g., by assuming that always the same average size object is consumed). This solution is simple, but asking each library to process every request would increase server-side work significantly. Alternatively, Popcorn could exploit layered coding [56, 63, 86, 92] or multiple description coding (MDC) [52, 89, 104]. There would be a single basic quality library accessed by all clients, with separate libraries for enhancement layers (better spatial resolution, bitrate, frame rate, etc.). Server-side work would thus be proportional only to the size of the highest quality library.

Billing and accounting. Popcorn must enable the content distributor to charge consumers, pay royalties, and collect aggregate statistics. Popcorn's current prototype can support both subscription-based and pay-per-view pricing models by monitoring accesses to the key server. Furthermore, it by default works with a prepaid royalty model, where the distributor pays a fixed license fee upfront. However, in its current form, Popcorn does not support advanced pricing models (different prices for different objects, possibly in tiers) or advanced royalties models (e.g., based on number of views or aggregate statistics). However, we believe that these limitations are not fundamental (e.g., prior works [14, 23, 55, 100] have addressed them in different contexts). Future work is to investigate how these prior works can be composed with Popcorn, and what the implications of doing so are on performance and privacy guarantees.

Targeted ads and recommendation services. Popcorn does not currently support targeted advertisements or recommendations, but we believe this limitation is not fundamental [17, 22, 24, 53, 62, 65]. Incorporating these results in Popcorn is a direction for future work.

[11] Assumes 10 billion hours watched in 3 months [4], requests are for a 90 minute video, and a total of 500 Points of Presence (PoP).

A Derivation of segment sizes

Recall the inequalities defined in §4.3:

$$t_i \le T_i \cdot \alpha, \quad \text{where } \alpha = \frac{\min\{R_i, P_i/b_i\}}{\mu \cdot n}$$

$$T_i \le d' + \sum_{j=1}^{i-1} t_j, \quad \text{where } d' = d - \epsilon$$

We consider the special case where both sides of the inequalities are equal. Combining both statements:

$$t_i = \left(d' + \sum_{j=1}^{i-1} t_j \right) \cdot \alpha$$

We show that $t_i = (d' \cdot \alpha \cdot (1+\alpha)^{i-1})$ is a solution to the above equation. Substituting on both sides:

$$d' \cdot \alpha \cdot (1+\alpha)^{i-1} = \left(d' + \sum_{j=1}^{i-1} d' \cdot \alpha \cdot (1+\alpha)^{j-1} \right) \cdot \alpha.$$

Summing the finite geometric series, and rearranging:

$$= \left(d' + d' \cdot \alpha \cdot \left(\frac{(1+\alpha)^{i-1} - 1}{\alpha} \right) \right) \cdot \alpha$$
$$= d' \cdot \alpha \cdot (1+\alpha)^{i-1}.$$

Setting $\alpha = 1$, we get $t_i = 2^{i-1} \cdot (d - \epsilon)$, as desired.

B Pricing model

Our high-level goal is to estimate the hourly cost of renting three resources on Amazon EC2: a vCPU, 1 GB of memory, and 1 Gbps of sequential read I/O bandwidth. To get the estimates, we make the simplifying assumption that the price of an EC2 machine depends only on these three resources. Of course, in practice, pricing machines is a complex process that depends on many factors (I/O performance for non-sequential workloads, cost of the networking infrastructure, prices set by competitors, etc.); the values derived here should be treated as only estimates.

At a high level, our method is to use the specification of machines on Amazon EC2 and their corresponding prices to derive a system of linear equations; in these equations variables represent the unit cost of the resources mentioned above, coefficients represent the "quantity" of those resources in an Amazon EC2 machine, and the RHS will be the price of renting that machine.

We consider the machines in Figure 8 and an additional machine. We need this additional machine as the equation for i2.4xl is not linearly independent from that of i2.8xl, which leaves us with two equations to solve for three variables. To write the third equation, we pick a memory optimized machine that has 32 vCPUs, 244 GB of memory capacity, 2 SSDs with 320 GB capacity each (6.4 Gbps sequential read I/O bandwidth), and is rented out for \$0.9822 per hour. Using these, we get the following equations:

$$\begin{aligned} 32C + 60M + 6.4I &= 0.6281 \\ 32C + 244M + 6.4I &= 0.9822 \\ 32C + 244M + 23.3I &= 1.6902, \end{aligned}$$

where C is the hourly cost of renting a vCPU, M is the cost of renting 1 GB of memory for an hour, and I is the hourly cost for 1 Gbps of sequential read I/O bandwidth.

Solving for the unknowns in the equations, we get I=0.042, M=0.0019, and C=0.0076.

Acknowledgments

We thank Carlos Aguilar-Melchor, Allen Clement, Alan Dunn, Bryan Ford, Yuval Ishai, Lon Ingram, Jaeyeon Jung, Brad Karp, Marc-Olivier Killijian, Sangman Kim, Michael Z. Lee, James Mickens, Thomas Schneider, Vitaly Shmatikov, Emmett Witchel, and the anonymous reviewers for feedback and comments that improved this draft. The Texas Advanced Computing Center (TACC) at UT supplied computing resources for an earlier version of this work. This work was supported by NSF grants 1040083, 1048269, 1409555, and 1055057; an AWS in Education Research grant ; and a Google European Doctoral Fellowship.

References

[1] Alphabetical List - Fri, Apr 3, 2015. `http://usa.netflixable.com/2015/04/alphabetical-list-fri-apr-3-2015.html`.

[2] Digital Rights Management. `http://msdn.microsoft.com/en-us/library/cc838192%28VS.95%29.aspx`.

[3] Microsoft PlayReady. `http://www.microsoft.com/playready/`.

[4] Netflix 2015 Q1 Earnings Letter. `http://files.shareholder.com/downloads/NFLX/47469957x0x821407/DB785B50-90FE-44DA-9F5B-37DBF0DCD0E1/Q1_15_Earnings_Letter_final_tables.pdf`.

[5] Netflix Soars On Subscriber Growth. `http://www.forbes.com/sites/laurengensler/2015/01/20/netflix-soars-on-subscriber-growth/`.

[6] New Movie Arrivals - Fri, Apr 3, 2015. `http://usa.netflixable.com/2015/04/new-movie-arrivals-fri-apr-3-2015.html`.

[7] The 2014 Pulitzer Prize Winners, Public Service: The Guardian US and The Washington Post. `http://www.pulitzer.org/works/2014-Public-Service`.

[8] The WebM Project. `http://www.webmproject.org/about/faq/`.

[9] WebM Encryption. `http://www.webmproject.org/docs/webm-encryption/`.

[10] You are watching more web video ads than ever. `http://allthingsd.com/20130215/you-are-watching-more-web-video-ads-than-ever/`.

[11] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang. Unreeling Netflix: Understanding and improving multi-CDN movie delivery. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2012.

[12] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private Information Retrieval for Everyone. Cryptology ePrint Archive, Report 2014/1025, 2014.

[13] C. Aguilar-Melchor and P. Gaborit. A lattice-based computationally-efficient private information retrieval protocol. In *Western European Workshop on Research in Cryptology (WEWoRC)*, 2007.

[14] W. Aiello, Y. Ishai, and O. Reingold. Priced oblivious transfer: How to sell digital goods. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2001.

[15] O. M. Alliance. DRM Architecture. `http://technical.openmobilealliance.org/Technical/release_program/docs/DRM/V2_1-20081106-A/OMA-AD-DRM-V2_1-20081014-A.pdf`, Mar. 2004.

[16] D. Asonov and J.-C. Freytag. Almost optimal private information retrieval. In *Workshop on Privacy Enhancing Technologies (PET)*, 2003.

[17] M. Backes, A. Kate, M. Maffei, and K. Pecina. ObliviAd: Provably secure and practical online behavioral advertising. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[18] E. Balsa, C. Troncoso, and C. Diaz. OB-PWS: Obfuscation-based private web search. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[19] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *Journal of Cryptology*, 17(2):125–151, 2004.

[20] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology—CRYPTO*, 2011.

[21] G. Brassard, C. Crepeau, and J.-M. Robert. All-or-nothing disclosure of secrets. In *Advances in Cryptology—CRYPTO*, 1987.

[22] M. Burkhart and X. A. Dimitropoulos. Fast privacy-preserving top-k queries using secret sharing. In *International Conference on Computer Communication Networks (ICCCN)*, 2010.

[23] J. Camenisch, M. Dubovitskaya, and G. Neven. Unlinkable priced oblivious transfer with rechargeable wallets. In *International Conference on Financial Cryptography and Data Security (FC)*, 2010.

[24] J. Canny. Collaborative filtering with privacy. In *IEEE Symposium on Security and Privacy (S&P)*, 2002.

[25] J. Cappos. Avoiding theoretical optimality to efficiently and privately retrieve security updates. In *International Conference on Financial Cryptography and Data Security (FC)*, 2013.

[26] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[27] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology—CRYPTO*, 2013.

[28] X. Cheng, C. Dale, and J. Liu. Statistics and social network of YouTube videos. In *International Workshop on Quality of Service (IWQoS)*, 2008.

[29] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[30] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280, 1990.

[31] W. Dai, Y. Doröz, and B. Sunar. Accelerating SWHE based PIRs using GPUs. Cryptology ePrint Archive, Report 2015/462, 2015.

[32] A. Davis, J. Parikh, and W. E. Weihl. Edgecomputing: Extending enterprise applications to the edge of the internet. In *International World Wide Web conference on Alternate track papers & posters (WWW Alt.)*, 2004.

[33] D. Demmler, A. Herzberg, and T. Schneider. RAID-PIR: Practical multi-server PIR. In *Cloud computing security workshop (CCSW)*, 2014.

[34] Y. G. Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–458, 1994.

[35] C. Devet. Evaluating private information retrieval on the cloud. Technical Report 5, University of Waterloo, 2013.

[36] C. Devet and I. Goldberg. The best of both worlds: Combining information-theoretic and computational PIR for communication efficiency. In *Privacy Enhancing Technologies Symposium (PETS)*, 2014.

[37] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *USENIX Security Symposium*, 2012.

[38] G. Di Crescenzo, T. Malkin, and R. Ostrovsky. Single database private information retrieval implies oblivious transfer. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2000.

[39] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[40] Discretix Technologies. Secure implementations of content protection DRM schemes on consumer electronic devices. `http://www.discretix.com/wp-content/uploads/2013/02/secure_implementation_of_content_protection_schemes_on_consumer_electronic_devices.pdf`, Feb. 2013.

[41] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. *ACM SIGCOMM Computer Communication Review*, 41(4):362–373, 2011.

[42] J. Domingo-Ferrer, A. Solanas, and J. Castellà-Roca. h(k)-private information retrieval from privacy-uncooperative queryable databases. *Online*

Information Review, 33(4):720–744, 2009.

[43] C. Dong and L. Chen. A fast single server private information retrieval protocol with low communication cost. In *European Symposium on Research in Computer Security (ESORICS)*, 2014.

[44] Electronic Frontier Foundation. NSA spying on Americans. `https://www.eff.org/nsa-spying`.

[45] W. Gasarch. A survey on private information retrieval. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 82:72–107, 2004.

[46] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC)*, 2009.

[47] Y. Gertner, S. Goldwasser, and T. Malkin. A random server model for private information retrieval. In *International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, 1998.

[48] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. In *ACM Symposium on Theory of Computing (STOC)*, 1998.

[49] I. Goldberg. Percy++ project on SourceForge. `http://percy.sourceforge.net/`.

[50] I. Goldberg. Improving the robustness of private information retrieval. In *IEEE Symposium on Security and Privacy (S&P)*, 2007.

[51] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[52] V. Goyal. Multiple description coding: compression meets the network. *IEEE Signal Processing Magazine*, 18(5):74–93, Sept. 2001.

[53] S. Guha, B. Cheng, and P. Francis. Privad: Practical privacy in online advertising. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[54] R. Henry, Y. Huang, and I. Goldberg. One (block) size fits all: PIR and SPIR over arbitrary-length records via multi-block PIR queries. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[55] R. Henry, F. Olumofin, and I. Goldberg. Practical PIR for electronic commerce. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[56] U. Horn, K. Stuhlmüller, M. Link, and B. Girod. Robust internet video transmission based on scalable coding and unequal error protection. *Signal Processing: Image Communication*, 15(1):77–94, 1999.

[57] Y. Huang and I. Goldberg. Outsourced private information retrieval. In *Workshop on Privacy in the Electronic Society (WPES)*, 2013.

[58] S. Huss-Lederman, E. M. Jacobson, J. Johnson, A. Tsao, and T. Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *ACM/IEEE Conference on Supercomputing*, 1996.

[59] A. Iliev and S. Smith. Private information storage with logarithmic-space secure hardware. In *Information Security Management, Education and Privacy*, 2004.

[60] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Batch codes and their applications. In *ACM Symposium on Theory of Computing (STOC)*, 2004.

[61] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[62] S. Jha, L. Kruger, and P. McDaniel. Privacy preserving clustering. In *European Symposium on Research in Computer Security (ESORICS)*, 2005.

[63] M. Johanson and A. Lie. Layered encoding and transmission of video in heterogeneous environments. In *ACM Multimedia (ACM-MM)*, 2002.

[64] J. Joskowicz and J. Ardao. Combining the effects of frame rate, bit rate, display size and video content in a parametric video quality model. In *Latin America Networking Conference (LANC)*, 2011.

[65] S. Katzenbeisser and M. Petković. Privacy-preserving recommendation systems for consumer healthcare services. In *International Conference on Availability, Reliability and Security (ARES)*, 2008.

[66] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Symposium on Foundations of Computer Science (FOCS)*, 1997.

[67] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Privacy Enhancing Technologies Symposium (PETS)*, 2016.

[68] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.

[69] M. Z. Lee, A. M. Dunn, B. Waters, E. Witchel, and J. Katz. Anon-pass: Practical anonymous subscriptions. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[70] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[71] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *International Conference on Financial Cryptography and Data Security (FC)*, 2015.

[72] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[73] T. Mayberry, E.-O. Blass, and A. H. Chan. PIRMAP: Efficient private information retrieval for MapReduce. In *International Conference on Financial Cryptography and Data Security (FC)*, 2013.

[74] T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining ORAM and PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[75] C. A. Melchor, B. Crespin, P. Gaborit, V. Jolivet, and P. Rousseau. High-speed private information retrieval computation on GPU. In *International Conference on Emerging Security Information, Systems and Technologies (SECUREWARE)*, 2008.

[76] M. Naor and B. Pinkas. Oblivious transfer and

polynomial evaluation. In *ACM Symposium on Theory of Computing (STOC)*, 1999.

[77] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[78] A. Narayanan and V. Shmatikov. Myths and fallacies of "personally identifiable information". *Communications of the ACM*, 53(6):24–26, June 2010.

[79] F. Olumofin and I. Goldberg. Preserving access privacy over large databases. Technical Report 33, University of Waterloo, 2010.

[80] F. Olumofin, P. K. Tysowski, I. Goldberg, and U. Hengartner. Achieving efficient query privacy for location based services. In *Privacy Enhancing Technologies Symposium (PETS)*, 2010.

[81] R. Ostrovsky and W. E. Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *Public Key Cryptography (PKC)*, 2007.

[82] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1999.

[83] S. Papadopoulos, S. Bakiras, and D. Papadias. pCloud: A distributed system for practical PIR. *IEEE Transactions on Dependable and Secure Computing*, 9(1):115–127, 2012.

[84] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind Seer: A scalable private DBMS. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[85] Pomelo LLC. Analysis of Netflix's security framework for 'Watch Instantly service. `http://pomelollc.files.wordpress.com/2009/04/pomelo-tech-report-netflix.pdf`, Mar. 2009.

[86] H. M. Radha, M. Van der Schaar, and Y. Chen. The MPEG-4 fine-grained scalable video coding method for multimedia streaming over IP. *IEEE Transactions on Multimedia*, 3(1):53–68, 2001.

[87] D. Rayburn. CDN market trends: Pricing, growth and competitive landscape. In *Content Delivery Summit*, 2015.

[88] D. Rebollo-Monedero and J. Forné. Optimized query forgery for private information retrieval. *IEEE Transactions on Information Theory*, 56(9):4631–4642, 2010.

[89] A. R. Reibman, H. Jafarkhani, Y. Wang, M. T. Orchard, and R. Puri. Multiple description coding for video using motion compensated prediction. In *International Conference on Image Processing (ICIP)*, 1999.

[90] F. Saint-Jean. Java implementation of a single-database computationally symmetric private information retrieval (cSPIR) protocol. Technical report, DTIC Document, 2005.

[91] B. Schneier. The eternal value of privacy. *Wired*, May 2006. `http://archive.wired.com/politics/security/commentary/securitymatters/2006/05/70886`.

[92] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the scalable video coding extension of the H.264/AVC standard. *Transactions on Circuits and Systems for Video Technology*, 17(9):1103–1120, 2007.

[93] R. Singel. Netflix spilled your *Brokeback Mountain* secret, lawsuit claims. *Wired*, Dec. 2009. `http://www.wired.com/images_blogs/threatlevel/2009/12/doe-v-netflix.pdf`.

[94] K. D. Singh, Y. Hadjadj-Aoul, and G. Rubino. Quality of experience estimation for adaptive HTTP/TCP video streaming using H.264/AVC. In *Consumer Communications and Networking Conference (CCNC)*, 2012.

[95] R. Sion and B. Carbunar. On the practicality of private information retrieval. In *Network and Distributed System Security Symposium (NDSS)*, Mar. 2007.

[96] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Systems Journal*, 40(3):683–695, 2001.

[97] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[98] J. P. Stern. A new and efficient all-or-nothing disclosure of secrets protocol. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 1998.

[99] J. Summers, T. Brecht, D. Eager, and B. Wong. Methodologies for generating HTTP streaming video workloads to evaluate web server performance. In *International Systems and Storage Conference (SYSTOR)*, 2012.

[100] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[101] J. Trostle and A. Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *International Security Conference (ISC)*, 2010.

[102] S. Viswanathan and T. Imielinski. Pyramid broadcasting for video-on-demand service. In *Multimedia Computing and Networking (MMCN)*, 1995.

[103] S. Wang, D. Agrawal, and A. El Abbadi. Generalizing PIR for practical private retrieval of public data. In *Working Conference on Data and Applications Security and Privacy (DBSec)*, 2010.

[104] Y. Wang and S. Lin. Error-resilient video coding using multiple description motion compensation. *IEEE Transactions on Circuits and Systems for Video Technology*, 12(6):438–452, 2002.

[105] P. Williams and R. Sion. Usable PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[106] S. Yekhanin. Private Information Retrieval. *Communications of the ACM*, 53(4):68–73, Apr. 2010.

[107] X. Yi, M. G. Kaosar, R. Paulet, and E. Bertino. Single-database private information retrieval from fully homomorphic encryption. *IEEE Transactions on Knowledge and Data Engineering*, 25(5):1125–1134, 2013.

[108] H. Yu, D. Zheng, B. Y. Zhao, and W. Zheng. Understanding user behavior in large-scale video-on-demand systems. *ACM SIGOPS Operating Systems Review*, 40(4):333–344, 2006.

Speeding up Web Page Loads with `Shandian`

Xiao Sophia Wang[*], *Arvind Krishnamurthy*[*], *and David Wetherall*[*†]

Abstract

Web page loads are slow due to intrinsic inefficiencies in the page load process. Our study shows that the inefficiencies are attributable not only to the contents and structure of the Web pages (e.g., three-fourths of the CSS resources are not used during the initial page load) but also the way that pages are loaded (e.g., 15% of page load times are spent waiting for parsing-blocking resources to be loaded).

To address these inefficiencies, this paper presents `Shandian` (which means lightening in Chinese) that restructures the page load process to speed up page loads. `Shandian` exercises control over what portions of the page gets communicated and in what order so that the initial page load is optimized. Unlike previous techniques, `Shandian` works on demand without requiring a training period, is compatible with existing latency-reducing techniques (e.g., caching and CDNs), supports security features that enforce same-origin policies, and does not impose additional privacy risks. Our evaluations show that `Shandian` reduces page load times by more than half for both mobile phones and desktops while incurring modest overheads to data usage.

1 Introduction

Web pages have become the de-facto standard for billions of users to get access to the Internet. The end-to-end Web page load time (PLT) has consequently become a key metric as it affects user experience and thus is associated with business revenues [6, 4]. Reports suggest that Shopzilla increased its revenue 12% by reducing PLT from 6 seconds to 1.2 seconds and that Amazon found every 100ms of increase in PLT cost them 1% in sales [27].

Despite its importance and various attempts to improve PLT, the end-to-end PLT for most pages is still a few seconds on desktops and more than ten seconds on mobile devices [9, 39]. This is because modern Web pages are often complex. Previous studies show that Web pages contain more than fifty Web objects on average [9], and exhibit complex inter-dependencies that result in inefficient utilization of network and compute resources [39]. In our own experiments, we have identified three types of inefficiencies associated with Web pages and the page load process. The first inefficiency comes from the content size of Web pages. Many Web pages use JavaScript libraries such as jQuery [21] or include large customized JavaScript code in order to support a high degree of user interactivity. The result is that a large portion of the code conveyed to a browser is never used on a page or is only used when a user triggers an action. The second inefficiency stems from how the different stages of the page load process are scheduled to ensure semantic correctness in the presence of concurrent access to shared resources. This results in limited overlap between computation and network transfer, thus increasing PLT. The third and related inefficiency is that many resources included in a Web page are often loaded sequentially due to the complex dependencies in the page load process, and this results in sub-optimal use of the network and increased PLTs.

Reducing PLT is hard given these inefficiencies. Human inspection is not ideal since there is no guarantee that Web developers adhere to the ever-changing best practices prescribed by experts [35]. Thus, it is widely believed that the inefficiencies should be transparently mitigated by automated tools and techniques. Many previously proposed techniques focus on improving the network transfer times. For example, techniques such as DNS pre-resolution [22], TCP pre-connect [19], and TCP fast open [28] reduce latencies, and the SPDY protocol improves network efficiency at the application layer [32]. Other techniques lower computation costs by either exploiting parallelism [25, 12] or adding software architecture support [41, 13]. While these techniques are moderately effective at speeding up the individual activities corresponding to a page load, they have had limited impact in reducing overall PLT, because they still communicate redundant code, stall in the presence of conflicting operations, and are constrained by the limited parallelism in the page load process.

The key and yet unresolved issue with page loads is that the page load process is suboptimally prioritized as to *what* portions of a page get loaded and *when*. In this paper, we advocate an approach that precisely prioritizes resources that are needed during the initial page load (load-time state) and those that are needed only after a page is loaded (post-load state). Unlike SPDY (or HTTP/2) server push and Klotski [10], which only prioritize network transfers at the granularity of Web objects, our approach prioritizes both network transfers and computation at a fine granularity (e.g., HTML elements and CSS rules), directly tackling the three inefficiencies listed above.

[*] University of Washington
[†] Google Inc.

A key challenge addressed by our approach is to ensure that we do not break static Web objects (e.g., external JavaScript and CSS), because caching and CDNs are commonly used to improve PLT. We make design decisions to send unmodified static contents in the post-load state thereby incurring the cost of sending a small portion of redundant content that is already included in the load-time state.

To deploy this approach transparently to Web pages, we choose a split-browser architecture and fulfill part of the page load on a proxy server, which can be either part of the web service itself (e.g., reverse proxies) or third-party proxy servers (e.g., Amazon EC2). A proxy server is set up to preload a Web page up to a time, e.g., when the *load* event is fired; the preload is expected to be fast since it exploits greater compute power at the proxy server and since all the resources that would normally result in blocking transfers are locally available. When migrating state (logics that determine a Web page and the stage of the page load process) to the client, the proxy server prioritizes state needed for the initial page load over state that will be used later, so as to convey critical information as fast as possible. After all the state is fully migrated, the user can interact with the page normally as if the page were loaded directly without using a proxy server.

Note that Opera mini [26] and Amazon Silk [3] also embrace a split-browser architecture but differ in terms of how the rendering process is split between the client and the proxy server. Their client-side browsers only handle display, and thus JavaScript evaluation is handled by the proxy server. This process depends on the network, which is both slow and unreliable in mobile settings [30], and encourages the proxy server to be placed near users. We have a fully functioning client-side browser and encourages the proxy server to be placed near front-end Web servers (e.g., edge POPs) for the most performance gains.

Our contributions are as follows:

- We conduct a measurement study that identifies the inefficiencies of Web pages that can be fixed by better page structures. We find that three-fourths of the CSS is not used during a page load and that parsing-blocking CSS and JavaScript slow down page loads by 20%.
- We design and implement Shandian, which significantly reduces end-to-end PLTs. Shandian uses a proxy server to preload a Web page, quickly communicates an initial representation of the page's DOM to the client, and loads secondary resources in the background. Shandian also ensures that the Web page functionality in terms of user interactivity is preserved and that the delivery process is compatible with latency-reducing techniques such as caching and CDNs and security features such as the enforcement of same-origin policies. The resulting system is thus both efficient and practical.
- We evaluate Shandian on the top 100 Alexa Web pages which have been heavily optimized by other technologies. Our evaluations still show that Shandian reduces PLT by more than half with a reasonably powerful proxy server on a variety of mobile settings with varied RTT, bandwidth, CPU power, and memory. For example, Shandian reduces PLT by 50% to 60% on a mobile phone with 1GHz CPU and 1GB memory by exploiting the compute power of a proxy server with a multicore 2.4GHz server. Unlike many techniques that only improve network or computation, Shandian shows consistent benefits on a variety of settings. We also find that the amount of load-time state is decreased while the total amount of traffic is increased moderately by 1%.

In the rest of this paper, we first review the background of Web pages and the page load process by identifying the inefficiencies associated with page loads (§2). Next, we present the design of Shandian (§3) and its implementations and deployment (§4). We evaluate Shandian in §5, discuss in §6, review related work in §7, and conclude in §8.

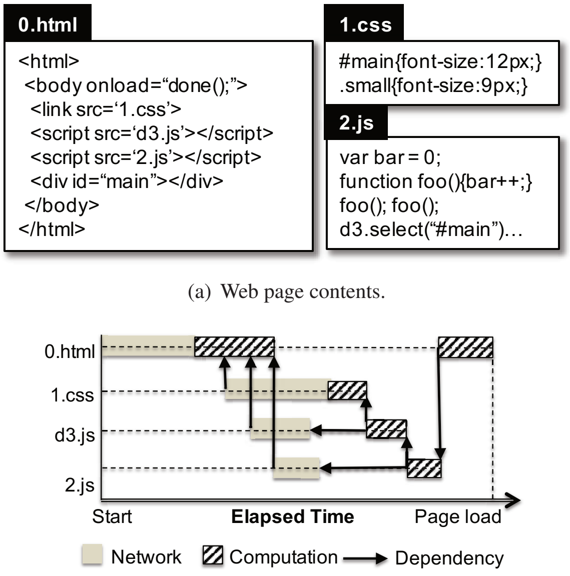
2 An analysis of page load inefficiencies

This section reviews the background on the Web page load process (§2.1), identifies three inefficiencies in the load process, and quantifies them using a measurement study (§2.2).

2.1 Background: Web page loads

Web page compositions. A Web page consists of several Web objects that can be HTML, JavaScript, CSS, images, and other media such as videos and Flash. HTML is the language (also the root object) that describes a Web page; it uses a markup to define a set of tree-structured elements such as headings, paragraphs, tables, and inputs. Cascading style sheets (CSS) are used for specifying presentation attributes such as colors and fonts of the HTML elements and is expressed as a set of rules. Processing the CSS involves identifying the HTML elements that match the given rules (referred to as CSS selector matching) and adding the specified styles to matched elements. JavaScript is often used to add dynamic content to Web pages; it can manipulate the HTML, say by adding new elements, modifying existing elements, or changing elements' styles, and can define and handle events. CSS and JavaScript are embedded in a Web page as HTML elements (i.e., script, style, and link) and can be either a standalone Web object or inline HTML.

Web page load process. First, when a user inputs or clicks a URL, the browser initiates an HTTP request to



0.html

```
<html>
 <body onload="done();">
  <link src='1.css'>
  <script src='d3.js'></script>
  <script src='2.js'></script>
  <div id="main"></div>
 </body>
</html>
```

1.css

```
#main{font-size:12px;}
.small{font-size:9px;}
```

2.js

```
var bar = 0;
function foo(){bar++;}
foo(); foo();
d3.select("#main")...
```

(a) Web page contents.

(b) Dependency graph of loading the page.

Figure 1: An example of loading a page.

that URL. Upon receiving the request, the server either responds with a static HTML file, or runs server-side code (e.g., Node.js or PHP) to generate the HTML contents on the fly and sends it to the browser. The browser then starts to parse the HTML contents; it downloads embedded files (e.g., CSS and JavaScript) until the page is fully parsed. The result of the parsing process is a document object model (DOM) tree, an in-memory representation of the Web page. The DOM tree provides a common interface for JavaScript to manipulate the page. The browser progressively renders the page during the load process; it converts the DOM tree to a layout tree and further to pixels on the screen.

The browser fires a `load` event when it finishes loading the DOM tree. The `load` event is commonly used for prioritizing Web page contents to improve user experience [9, 39]. For example, websites commonly use the `load` event to defer loading JavaScript that is not used in the initial page display. Such a design makes Web pages more responsive and provides better user-perceived page load times.

Dependencies in Web page loads. Ideally, the browser should fetch Web objects of a page fully in parallel, but in practice the process is often blocked by dependencies among Web objects. One type of dependencies stems from coordinating access to shared resources [39]. For example in Figure 1, when the parser encounters the `script` tag that references 2.js, it stops parsing, loads the corresponding JavaScript, evaluates the script (i.e., compilation and execution), and then resumes parsing. As both HTML and JavaScript can modify the DOM, this ensures that the DOM is modified in the order specified in the Web page. When a CSS appears ahead of this JavaScript (e.g., 1.css), evaluating the JavaScript needs to wait until the CSS is loaded and evaluated. (CSS evaluation includes parsing CSS rules, matching CSS selectors, and computing element styles.) This is because both JavaScript and CSS can modify the elements' styles in the DOM. As is shown in Figure 1(b), HTML parsing is often blocked to ensure the correctness of execution, thus significantly slowing down page loads.[1]

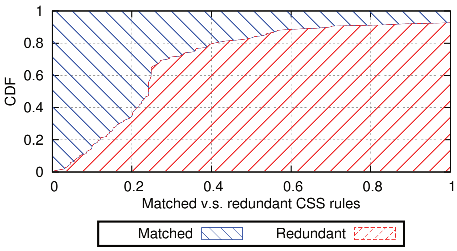


Figure 2: Fraction of redundant CSS rules of top 100 pages in bytes.

Unlike CSS and JavaScript, other Web objects (e.g., images) do not block HTML parsing or any task other than rendering. Therefore, the composition of HTML, CSS, and JavaScript resources and how they are organized are often the factors that affect PLT.

The dependencies not only slow down page loads but also prevent optimizations from being more effective. For example, the SPDY protocol would significantly improve PLT if all the objects in a page were fetched and processed in parallel; but this improvement is largely nullified by the page dependencies in real browser contexts [40]. This is because the optimization technologies often just improve one aspect of page loads (e.g., network utilization), but the overall page load process remains constrained by dependencies and the marginal improvements are not significant.

Critical paths of Web page loads. Not all the object loads on a Web page affect the PLT. The bottlenecks can be identified by performing a *critical path analysis* on the dependency graphs obtained when a page is being loaded. For example in Figure 1(b), loading 0.html and 1.css and evaluating all the objects are on the critical path. Figure 1(b) shows that the time spent on the network comes not only from time to load the HTML, but also from blocking load of JavaScript or CSS (e.g., 1.css), which significantly slows down PLT.

2.2 Page load inefficiencies

To understand inefficiencies in the Web page load process, we conduct a study on top 100 Alexa [2] pages by using Chrome (which is a highly optimized browser). To

[1] These dependencies are enforced by popular browsers including Chrome, Firefox, Safari, and IE.

provide a controlled network environment, we download all pages to our own server, and use Dummynet [11] to provide a stable network connection of 20ms RTT and 10Mbps bandwidth. Our client is a machine with a 2GHz dual core CPU and 4GB memory. We clear the cache for every page load, and define PLT as the time between when the page is requested and when the `load` event is fired. We then identified the following factors that slow down the page load.

Unused CSS in page loads. The first observation is that CSS files often contain rules that are either *never* used in a page or at least *not* used during the initial page load. Such CSS rules incur unnecessary network traffic and parsing efforts. We quantify the amount of used versus unused CSS rules in initial page loads (see Figure 2). In particular, 75% of CSS rules are unused in the median case. Surprisingly, 80% and 96% of CSS rules are unused for `google.com` and `facebook.com` respectively. This suggests that CSS is likely to be redundant for interactive pages, because interactive pages tend to load lots of CSS rules for future interactions, at the cost of increased PLT.

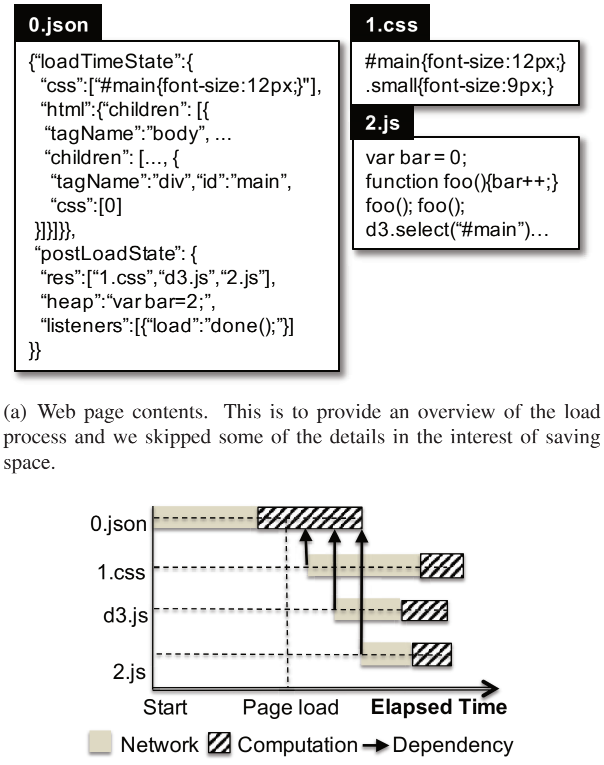
Blocking JS/CSS. JavaScript and CSS often block parsing on the critical path. We extend WProf [39] to measure the amount of additional round trips and parsing-blocking object downloads and evaluations. We find that 15% of the PLT for top pages is spent waiting for JavaScript or CSS to be loaded on the critical path, and 5% of PLT is used for evaluating CSS and JavaScript. Compared to the time to first byte, which is difficult to reduce, there are significant potential gains from optimizing CSS and JavaScript.

Additional round trips. Web objects are not loaded in a batch, but are often loaded sequentially due to the above reason. The result is that loading a page usually incurs many round trips, since loading an object often triggers a sequence of latency-inflating operations such as redirections, DNS lookups, TCP connection setups, and SSL handshakes. We find that 80% of pages have sequentially loaded Web objects on the critical path.

3 Design

Our design aims to reduce PLTs by restructuring the page load process to remove the inefficiencies measured in §2.2. We pre-process Web pages on the proxy server and migrate page state to the client in order to streamline the client-side page load process.

The key to our design is the state that we capture and migrate. On the one hand, page state needs to be captured at an appropriate processing stage in order to minimize the network and computational costs; on the other hand, the captured state should be comprehensive and ensure that the rendered page on the client displays and functions correctly. Figure 3 shows an example of loading a page with `Shandian`. We reorganize the state in the root object (e.g., 0.json) while keeping the integrity of other objects (e.g., 1.css, d3.js, and 2.js). Figure 3(b) shows a dependency graph of loading the page with `Shandian`. The page is loaded when the load-time state is loaded and evaluated (e.g., the `#main` element is rendered), which is much faster than Figure 1(b) with object inter-dependencies. Processing post-load state does not involve any complex inter-dependencies as the evaluation of objects can happen in parallel.



0.json

```
{“loadTimeState”:{
  “css”:[“#main{font-size:12px;}”],
  “html”:{“children”: [{
   “tagName”:”body”, ...
   “children”: [..., {
    “tagName”:”div”,“id”:”main”,
    “css”:[0]
 }]}]}},
 “postLoadState”: {
  “res”:[“1.css”,“d3.js”,“2.js”],
  “heap”:“var bar=2;”,
  “listeners”:[{“load”:”done();”}]
}}
```

1.css

```
#main{font-size:12px;}
.small{font-size:9px;}
```

2.js

```
var bar = 0;
function foo(){bar++;}
foo(); foo();
d3.select(“#main”)...
```

(a) Web page contents. This is to provide an overview of the load process and we skipped some of the details in the interest of saving space.

(b) Dependency graph of loading the page.

Figure 3: An example of loading a page using `Shandian`.

The challenges are detailed in §3.1. Next, we describe the *load-time state* (§3.2) that is captured for fast page loads, and the *post-load state* (§3.3) that is captured for interactivity and compatibility. In addition to the state that is migrated from the server to the client, we discuss the state that needs to be migrated from the client to the server (§3.4).

3.1 Challenges

We identify three challenges in designing `Shandian`.

First, precisely identifying state that is needed during a page load (load-time state) is nontrivial since load-time state and post-load state are largely mingled. For example, some Web pages use a small portion of jQuery [21] to construct HTML elements while leaving a large portion of jQuery unused. Precisely identifying the load-time state and migrating them to the client in the first place is key to reducing PLT.

Second, we need to ensure that the Web page rendered using Shandian is functionally equivalent to one that is computed solely on the client. On the one hand, as page state processing happens on both proxy servers and clients, we need to carefully design the split process to ensure that we do not break the pages. On the other hand, the server needs proper client-side state to function properly. For example, some Web pages that adopt a responsive Web design provide layouts specific to browser size. The server needs information about browser size, cookies or HTML5 local storage in order to function properly.

Third, completely recording and migrating the Web page state computed by the server is nontrivial. After the initial load process, the state computed by the server is largely dispersed across various JavaScript code fragments that comprise the Web page. This state needs to be retrieved and then migrated to their equivalent locations on the client in order to ensure that the user has a seamless experience in interacting with the Web page.

In the rest of the section, we discuss how we address these challenges in designing Shandian.

3.2 Load-time state

The load-time state is designed primarily for facilitating display and is captured at a processing stage that minimizes the amount of work required for rendering on the client. To this end, design decisions regarding the load-time state focus on *how much we can eliminate JavaScript/CSS evaluations while keeping the communicated state small*. As a result, the load-time state contains only HTML elements and their styles, but not JavaScript or post-load CSS. Below, we explain this in greater detail and also describe the state that is migrated to reflect JavaScript/CSS evaluations performed at the server.

3.2.1 Load-time state in JavaScript

JavaScript itself does not directly reflect on display, but the result of JavaScript evaluation can. As JavaScript evaluation is slow and blocks rendering, a design decision is to avoid both communicating JavaScript as part of the load-time state and evaluating JavaScript on the client device. Instead, the load-time state includes the result of JavaScript evaluation on the server, and this result is reflected in the HTML elements and their styles. For example, instead of transmitting a piece of D3 JavaScript [14] to construct an SVG graphic on the client, the JavaScript is evaluated at the server to generate the load-time state of HTML elements that represent the SVG. This design minimizes the computation time used in JavaScript evaluation for an initial page load and also avoids blocking executions on the client, but this is at the cost of potentially increased size of migrated state.

3.2.2 Load-time state in CSS

CSS evaluation is also slow, blocks rendering, and thus should be avoided in the initial page load as much as possible. The result of CSS evaluation is, however, a detailed and potentially unwieldy list of styles for each HTML element. Including the detailed list of styles in the load-time state would fully eliminate the CSS evaluation but incur a significant amount of time transferring the state.

Here, we seek an intermediate representation for the CSS state that incurs little additional time to finish the CSS evaluation while keeping the state small. CSS evaluation involves a sequential process of CSS parsing, CSS selector matching, and style computation. The CSS selector matching step matches the selectors of *all* the CSS rules to *each* HTML element, requiring more than a linear amount of time. The style computation process applies matched CSS properties in a proper order to generate a list of styles for rendering.

Our design decision here is to perform CSS parsing and matching on the server but leave style computations to be performed on the client. We migrate all the inputs required by style computations as part of load-time state. The required inputs are largely determined by the W3C algorithm that specifies the order according to which CSS properties are applied [37]. In addition to matched CSS selectors and properties for a given HTML element, the state also includes the importance (marked as important or not), the origin (from website, device, or user), and the specificity (calculated from CSS selectors) that determines this order. The resulting migrated state is compact compared to the detailed list of styles, and at the same time, it eliminates CSS selector matching on the client.

3.2.3 Serialization and deserialization

In order to obtain the load-time state, the proxy server first loads a Web page using a browser that has sufficient capabilities to handle HTML, CSS, and JavaScript. When the page load event or any other defined event is fired, the proxy browser serializes the load-time state from the memory. The server-side Shandian recursively serializes each HTML element in the DOM (excluding CSS and JavaScript elements), its attributes, and references to matched CSS rules. Then, the details of the matched CSS rules are serialized. Each CSS rule includes a CSS selector, a list of CSS properties, the importance, and the origin. Note that we do not add CSS rules to each matched HTML element, but use references to link HTML elements to their matched CSS rules (e.g., Figure 3(a) references the index of the matched CSS rule in the CSS array). This is because a CSS rule is likely to match with many HTML elements. The HTML elements and matched CSS rules together provide complete information for a page to be displayed properly on the client.

Deserializing the load-time state, which is both simple and fast, determines the page load time on the client. The client-side Shandian linearly scans the load-time state and uses HTML elements and attributes to construct the DOM. Instead of running a full CSS evaluation, Shandian computes styles from already matched CSS, requiring just a linear amount of time. Shandian does not require any client-side JavaScript evaluations because the state already contains the results of JavaScript evaluation. Compared to the page load process, the deserialization process does not block, does not incur additional network interactions, and avoids parsing of unused CSS or JavaScript, thereby significantly speeding up page loads on the client as is demonstrated by Figure 3(b).

3.3 Post-load state

Following the load-time state, the post-load state needs to be processed transparently in the background in order to ensure that: (a) users can further interact with the page as if it were delivered normally and not through Shandian (*interactivity*), and (b) latency-reduction techniques are still viable (*compatibility*). To ensure interactivity, the post-load state should include the portion of JavaScript that was not used in the load-time state, together with unused CSS, because they might be required later in user interactions. To ensure that complementary latency-reduction techniques such as caching and CDNs can be used in Shandian, we need to an unmodified version of external objects in the post-load state.

The most direct approach would be to migrate unmodified JavaScript/CSS snippets, which both ensures integrity (and thus compatibility) and includes all the information for post-load state (interactivity). Our design here focuses on examining the feasibility of migrating unmodified snippets and processing unmodified snippets while excluding the effects of load-time state.

3.3.1 Post-load state in CSS

Attaching unmodified CSS snippets (copies of inline CSS and links to external CSS) in the post-load state is both feasible and simple. We can just evaluate all the CSS rules here regardless of whether they had appeared in the load-time state. This is because CSS evaluation is idempotent—evaluating the same CSS rule any number of times would give the same results. In our design, the CSS rules in load-time state will be evaluated twice (one on the proxy server, and the other on the client) while post-load CSS is evaluated once.

This design is simple and satisfies the constraints, but at the cost of repeating the evaluation of load-time state. For example, if a snippet of external CSS is already being cached, our design does not require loading any portion of this snippet from anywhere else. The price to pay is the additional energy consumption and latencies that result from the repeated evaluation of load-time state. But, since these computations happen after the initial load-time version has been rendered, the additional cost does not impact user's perception of the page load time.

3.3.2 Post-load state in JavaScript

Attaching unmodified JavaScript snippets in the post-load state incurs a complex processing procedure, because not all JavaScript evaluation is idempotent. On the one hand, we need to ensure that JavaScript evaluation has equivalent results as if Shandian weren't used; on the other hand, we need to completely record all the state of JavaScript. Other approaches such as migrating the entire heap would incur significantly larger state (10x) and break the integrity of JavaScript objects (consequently caching), and are thus not an option here.

Ensuring equivalent results from JavaScript evaluations. If we include the original unmodified JavaScript code in the post-load state, it is hard to ensure that JavaScript evaluation gives equivalent results as if Shandian weren't used. This is because the order in which JavaScript appears determines the results of JavaScript evaluation, but unfortunately this order is not preserved as a result of isolating load-time and post-load state. If we do not keep unmodified JavaScript in the post-load state, the compatibility would be compromised, so is the size of the communicated state.

Our approach uses unmodified JavaScript, together with a bit of the memory state that we call heap (referred to as partial heap), to reconstruct the whole heap. To keep the partial heap small, the key is to extract as much information as possible from the unmodified JavaScript.

To this end, we further break down the unmodified JavaScript snippets into statements, and reuse as many idempotent statements as possible. A JavaScript statement can be a function declaration, a function call, a variable declaration, and so forth. Evaluating function declarations is idempotent, but evaluating other statements is not necessarily idempotent. To avoid double evaluating non-idempotent statements, the client-side Shandian only evaluates function declarations in post-load JavaScript, and directly applies the partial heap—the results of JavaScript evaluation that are migrated from the proxy server.

The contents of the partial heap largely depend on how function declarations would be extracted from unmodified JavaScript. However, isolating function declarations from other JavaScript is nontrivial because they are often largely mixed. Below, we discuss the situations under

Events	Event state
DOM events	event name, callback and its arguments
XmlHttpRequest	internal fields of the object
setTimeout	time to fire, callback and its arguments
setInterval	time to fire, interval, callback and its arguments

Table 1: Summary of events and their states.

which function declarations are hard to isolate and also describe the use of the partial heap when necessary.

(i) Recursively embedded instance variables. JavaScript does not distinguish between functions and objects, and thus a function declaration can recursively embed other function declarations and instance variables. To this end, the server-side Shandian recursively captures all the instance variables as the partial heap even if they are embedded in a function declaration. When the client-side Shandian evaluates unmodified JavaScript, it first only evaluates function declarations by ignoring these instance variables, and then applies the partial heap to restore the instance variables.

(ii) Self-invoking functions. A self-invoking function combines a function declaration and a function call in a single statement. For example, `(function(n) {alert(n);})(0)` is a self-invoking function. Our approach is to split up the single statement into a function declaration and a function call, and evaluate them differently.

(iii) `eval` and `document.write` can convert strings to JavaScript code that embeds function declarations. The use of `eval` and `document.write` is considered as bad practices for both performance and security. We disable the use of Shandian for Web pages that have invoked `eval` and `document.write` before a page is loaded.

Recording all the state of JavaScript. Recording all the state is challenging, because some state such as those in function closures and event callbacks are hard to capture.

(i) Instance variables in function closures. A function closure is often used for isolating code execution environments (referred to as scopes). We instrument the JavaScript engine with the ability to refer to function closures and serialize the instance variable for each closure respectively. Unlike other techniques that handle function closures by rewriting JavaScript [23], instrumenting the JavaScript engine allows us to handle function closures efficiently.

(ii) State in event callbacks. Besides function closures, event callbacks are also hard to capture. Here, we consider three kinds of events that can be added in an initial page load, which are summarized in Table 1. Serializing the event callbacks requires us to capture all the state in the event queue.

3.3.3 Serialization and deserialization

Serialization and deserialization of the post-load state happens in the background while users interact with load-time state and is more complex than that of the load-time state.

The server-side Shandian first serializes unmodified CSS or JavaScript snippets if they are inline (their links instead if they are external), ensuring compatibility. Next, Shandian serializes the event callbacks and the partial heap excluding those that can be restored from function declarations in the unmodified JavaScript (e.g., `listeners` and `heap` fields in Figure 3(a)). The post-load state together with load-time state provides complete information for a Web page to function correctly. Note that the size of the load-time and post-load state together exceeds that of the original Web page. The extra portions include the matched CSS rules, the partial heap, and event state. Because they are computed from the original Web page and are thus repetitive, they can be compressed.

The client-side Shandian first deserializes and parses unmodified CSS and JavaScript, fetching corresponding objects if they are external. Unlike in a Web page where fetching CSS and JavaScript has to comply with the dependency model [39], here JavaScript and CSS objects can be fetched completely in parallel. After all the objects are fetched, CSS is completely evaluated, and the function declarations in JavaScript are evaluated to avoid duplicate evaluations. Then, the partial heap is applied and events start to get fired. At this point, the Web page state on the client is restored as if the entire page load process happened on the client.

3.4 Client-side state

Website information stored in browsers. In addition to migrating state from the server to the client, some state stored in browsers needs to be first migrated from the client to the server. While constructing the DOM, the browser uses long-term storage including cookies, HTML5 local storage, and Web database. Because the server does not keep a copy of this state, lacking client-side state might break the Web page. Shandian handles client-side state by migrating them from the client to the server along with the page request. But this has the potential to increase the uplink transfers and thus slow down page loads. To this end, we conduct a measurement study on client-side state and have confirmed in §5.4 that the client-side state that needs to be migrated is small.

Other sources of inconsistencies. Besides browser storage, there can be differences in obtaining timestamps (`Date.now`), geolocation, and browser information from the client and the server [8]. The absolute timestamps should be the same on the client and the server as if they are both synchronized to the global clock. However, the time zone could be different and thus needs to be sent to the server. The geolocation can only be obtained by asking users for an explicit consent. Once this happens, we send the geolocation to the server. Browser information includes the window size and user agents. As user agents are always sent to the server, we do not need to explicitly handle it. We send the window size to the server because it can be used to adjust the size of the layout (e.g., in a responsive design).

4 Deployment and Implementation

4.1 Deployment

Shandian can be deployed either in the reverse proxy (co-located with front-end servers) or as a separate globally distributed proxy service (similar to Opera mini [26] and Amazon Silk [3]).

Conventional wisdom suggests deployments near clients in order to make better use of edge caching and CDNs and to offer low latencies (e.g., when JavaScript offloading is needed). To the best of our knowledge, all page rewriting techniques (e.g., Opera Mini [26] and Amazon Silk [3]) are intended to be deployed near clients. Unfortunately, such a deployment slows down the preload process on the proxy server, because it adds additional round trips to the Web server, which is a key inefficiency especially for parsing-blocking object downloads in current Web pages (§2.2).

In our design of Shandian, we find that exploiting caching/CDNs and reducing round trip delays to the origin server are not at odds and that a carefully designed system can achieve both. We only require the resources that are used as part of the initial page load to go through the proxy server, while the resources accessed after the initial load (e.g., images and videos) can still be cached or be fetched from CDNs. Therefore, we consider deployments wherein the proxy server is located near the Web content server and is ideally co-located with the reverse proxy of the Web service in order to reduce the preprocessing time in the proxy server.

4.2 Implementation

State format. We represent the migrated state in JSON format, because it is simple and compact. Note that other formats such as XML or HTML are also viable.

Server-side Shandian. We implement the server-side browser as a webserver extension based on Chrome's content_shell with most modifications to Blink and few to V8. We chose the lightweight browser content_shell because it includes only page-specific features such as HTML5 and GPU acceleration, but not browser-specific features such as extensions, autofill, and spell checking [18].

Our instrumentation is primarily for state serialization, and is minimal before state serialization starts: we turn off downloads of images and other media because they are not part of the migrated state; we also block objects that are hosted on other domains because we mandate downloading all the required CSS and JavaScript to the Web server. While most of the state resides in Blink [5], some also resides in V8 [36] (e.g., event callbacks and function closures). The server extension can be added to any webserver that allows process invocation.

Client-side Shandian. The client-side browser is also based on Chrome, and we modify it as little as possible. We implemented a JSON lexer to parse the migrated state, and this lexer is invoked instead of the HTML lexer. After obtaining the HTML elements from the JSON lexer, we perform DOM construction using unmodified Blink. Given that the migrated state contains matched CSS rules, we skip CSS selector matching and directly apply the CSS properties to compute the element styles. We modify V8 a little to selectively evaluate function statements in JavaScript and to apply server-side results of JavaScript evaluations. We modify Blink to create event listeners and timers from our serialized state instead of executing the load-time JavaScript. The client-side browser can opt in to using Shandian using an HTTP header and thus can easily fallback to loading the original pages that Shandian does not support.

Note that we chose to modify the browsers instead of implementing state migration using JavaScript because JavaScript evaluation is time consuming and because it does not provide the appropriate APIs necessary for all the low-level manipulations. For example, JavaScript does not allow access to the matched CSS rules for an HTML element. By operating inside the browser code base, we have easy in-memory access to all the desired information, and we also avoid JavaScript execution at the client prior to the page load event.

5 Evaluation

The evaluation aims to demonstrate that: (i) Shandian significantly improves PLT under a variety of scenarios (§5.2), (ii) Shandian does not significantly hurt data usage §5.3), and (iii) the amount of client-side state that needs to be transferred to the server is small (§5.4).

5.1 Experimental setup

We conduct the experiments by setting up a client that loads Web pages using our modified Chrome and a server that hosts pages using our server extension. We detail the experimental setup below.

Our server is a 64-bit machine with 2.4GHz 16 core CPU and 16GB memory, and has an Ubuntu 12.04 installation with the 3.8.0-29 kernel. To ensure all the Web objects are co-located with Shandian, we download the mobile home pages of the Alexa top 100 websites to our server and use Apache to host them. We download all the Web objects for a page, ensuring that the page loads by our server extension do not issue external network requests. In practice, only Web objects that are used in initial page loads need to be hosted on the server. For example, synchronous JavaScript needs to be placed on the server, but images and videos can be placed anywhere else. Our experimental setup emulates a deployment setting where Shandian executes on a front-end server of a Web service.

Our clients include mobile phones (Nexus S with 1GHz Cortex A8 CPU, 512MB RAM) and virtual machines with varying CPU and memory. We experimented with a 3G/4G cellular network, WiFi, and Ethernet in a LAN. We focus on the results from LAN because results from the cellular network are similar to simulated LAN settings.

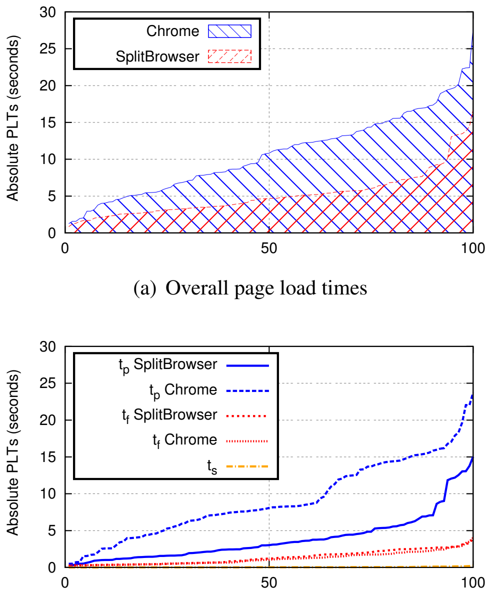
We define page load time (PLT) as the time to display page contents that are rendered before the W3C `load` event [38] is fired. Note that our approach works with any metrics of page load times, though we use the W3C `load` metric to evaluate our prototype. Alternatives to PLT such as the above-the-fold time (AFT) [7] and speed index [34] represent user-perceived page load times, but they require cumbersome video recordings and analysis and are thus out of the scope of this paper. We clear browser cache between any two page loads and do not consider client-side state that requires a login. We report the median page load times out of five runs for all the experiments.

5.2 Page load times

One source of benefits of Shandian comes from reducing the dependencies between network and computation, which in turn eliminates network operations that block rendering. For example in Figure 3(b), the network interaction is minimized to just fetching the load-time state in 0.json (before the page is loaded). Another source of benefits comes from reducing the amount of computation needed for the initial page loads (evaluating just the load-time state instead of evaluating all the dependent resources). We now demonstrate the performance benefits under a wide variety of scenarios.

5.2.1 PLT on mobile devices

We use a mobile phone, Nexus S with 1GHz Cortex-A8 CPU, 16GB internal memory (512MB RAM), and Android 4.1.2. The mobile phone is connected to the Internet via WiFi. We install our modified Android Chrome and automate experiments using `adb shell`. We load the Web pages with Shandian and with unmodified Chrome on the mobile phone. Figure 4(a) shows that the PLTs with Shandian are significantly reduced compared to those with unmodified Chrome. The reduction is as much as 60% in the median case.



(a) Overall page load times

(b) Breakdowns of page load times

Figure 4: Page load times (seconds) on Nexus S with 1GHz Cortex A8 CPU, 512MB RAM, and Android 4.1.2, and with WiFi. Shandian reduces page load times by 60% compared to Chrome in the median case.

Source of benefits: To identify the source of benefits, we further break down PLTs into time spent by the Shandian server extension t_s, time to fetch the first chunk of the page t_f, and time to parse the page (including parsing-blocking network fetch time) t_p. Figure 4(b) shows that Shandian's server extension uses little time to pre-process pages (22ms in the median case, and 250 ms in the maximum case). Compared to client-side page loads that take a few seconds, server-side page loads have negligible overheads, due to the benefits accrued from more compute power (especially memory), a lightweight server browser, and mitigated network inefficiencies by deploying the cloud server near the Web server. The benefits together suggest that migrating page load computations to the server is effective. By comparing the client-side parsing times t_p of Shandian and Chrome, we find that the benefits of Shandian stem mainly from client-side parsing. This is because Shandian requires no JavaScript evaluations, eliminates redundant CSS, and increases network utilization by eliminating blocking operations.

5.2.2 PLT on desktop VM

To demonstrate how much Shandian helps PLTs on a variety of scenarios, we use a desktop VM with Ubuntu 12.04 kernel 3.8.0-29 installed and connected to the net-

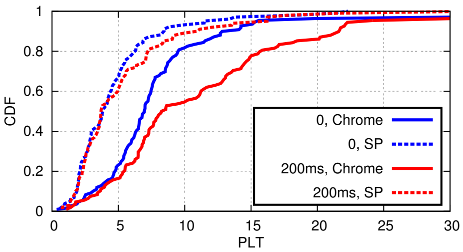


Figure 5: Varying RTT with fixed 1GHz CPU and 1GB memory.

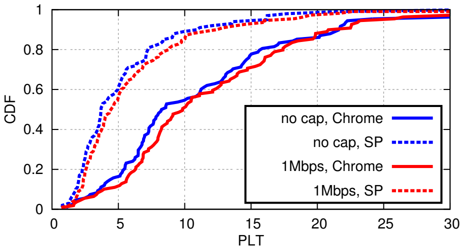


Figure 6: Varying bandwidth with fixed 1GHz CPU, 1GB memory, and 200ms RTT.

work using Ethernet. We use Dummynet [11] to emulate varying bandwidths and RTTs.

Varying RTT: We vary RTT from the minimal of the LAN to 200 milliseconds with fixed 1GHz CPU and 1GB memory, which are representative of current mobile devices. We do not cap the bandwidth. The scenario of minimal RTT approximates the scenario of having caching always enabled. Figure 5 shows the cumulative distributions of PLTs of the 100 Web pages. The increased RTT affects much of PLT with Chrome but affects little of PLT with Shandian, meaning that Shandian is insensitive to RTT. This is because among the breakdowns of PLT only t_f which is a small fraction of PLT is affected by RTT.

Varying bandwidth: We experiment with a 1Mbps bandwidth and with no bandwidth cap using fixed 1GHz CPU, 1GB memory, and 200ms RTT. Figure 6 shows that PLTs are affected little by bandwidths, which is consistent with previous findings [33, 31] that bandwidth is not a limiting factor of PLTs. We also run experiments in a cellular network but find similar results to simulated links.

Varying CPU: We vary CPU speed from 1GHz to 2GHz while fixing memory size to 1GB. We do not tune RTT or bandwidth, meaning that PLT is dominated by computation. Figure 7 shows that the PLT improvement is linear to CPU increase. It also shows that CPU speed has the same amount of impact for both Shandian and Chrome, because processing load-time state in Shandian still incurs lots of CPU cycles. As PLT is dominated by computation, the results here approximate the situations when objects are inlined or cached. Clearly, Shandian significantly improves PLTs than simply inlining objects since JavaScript evaluations and most of CSS evaluations are removed from the page load process.

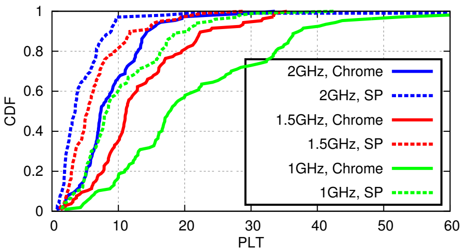


Figure 7: Varying CPU speed with fixed 1GB memory, no bandwidth cap, and no RTT insertion.

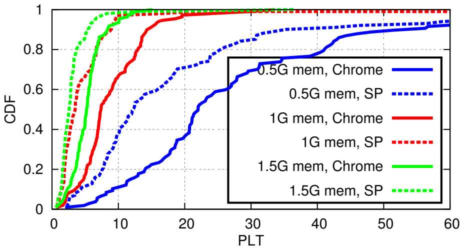


Figure 8: Varying memory size with fixed 2GHz CPU, no bandwidth cap, and no RTT insertion.

Varying memory: We vary memory size from 0.5GB to 1.5GB with fixed 1GHz CPU and no network tuning. Figure 8 suggests that memory size has the same amount of impact for both Shandian and Chrome, but a decrease in memory size has a more than linear negative impact on PLT.

In summary, Shandian significantly improves PLT compared to Chrome under a variety of realistic mobile scenarios. This is rare since most techniques are specific to improve one of computation and network. But Shandian improves both.

Note that we do not evaluate page interactivity metrics, e.g., the time until interaction is possible, because users spend time on the contents of a Web page before interacting with it and it is difficult to model this delay. Shandian could improve the time until interaction since all external resources are loaded and evaluated in parallel, but it can also hurt if the load-time state is too large and blocks the transfer of the post-load state that is required for page interactivity.

5.3 Size of transferred data

We evaluate the transferred data size as to (i) whether it hurts latencies and (ii) whether it hurts data usage. To understand whether the size of transferred data helps

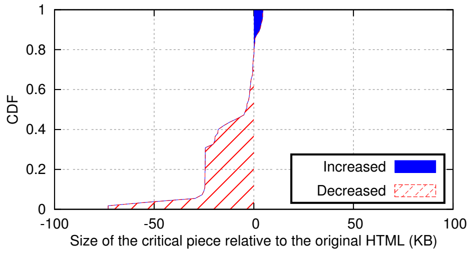


Figure 9: Size of the critical piece relative to the original HTML (KB).

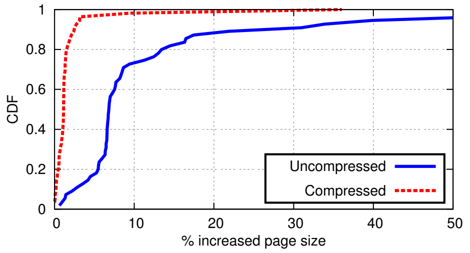


Figure 10: Percentage of increased page size (uncompressed v.s. compressed).

or hurts latencies, we consider the size of the load-time state, because the post-load state and other objects do not affect the page load time. Figure 9 shows the size of the load-time state when a standard gzip compression is applied. The size of the load-time state relative to the original HTML decreases for most pages, and increases by a small amount only for less than 20% of the pages. This means that our migrated load-time state improves latencies in overall.

To evaluate whether the size of migrated state hurts data usage which is important for mobile browsers, we consider the total size of Web pages transferred to the client including all the embedded objects. Figure 10 shows that the transferred data increases by 7% before compression, but it drops to 1% with standard gzip compression. This indicates that the overheads introduced by our approach are minimal.

5.4 Client-side state

We obtain client-side state from the browsers of a group of people, totaling 2,435 domains. The majority of the client-side state is HTML5 localStorage and cookies. We find that 90% of the websites use less than 460 bytes of localStorage, while 2% of the websites use more than 100KB of localStorage. Large localStorage is almost always used for caching. For example, websites that use CloudFlare keep many caches in their localStorage; social networking websites such as Facebook store friends lists in the localStorage; and location-based services maintain the points of interest in the localStorage. Lack of such localStorage does not break Web pages because cache misses can be remedied by fetching from the server. Unlike localStorage, cookies are widely used but are always small, and Web databases are used in less than 1% of the websites and are small. The use of sessionStorage is also not much, likely because it only persists per-session state and cannot cache as long as localStorage. The study of client-side state suggests that migrating all the necessary client-side state (e.g., cookies) to the server has a negligible effect on page load time.

6 Discussion

We believe that Shandian is an important first step in mitigating dependencies that are the key bottleneck of latencies in page loads. While future advances in JavaScript or the Web might require us to patch Shandian so as to ensure that Shandian does not break Web pages, there are no fundamental obstacles that prevent us from patching Shandian to track changes to the web page formats. Next, we discuss privacy, compatibility, and further optimizations that can be added to Shandian.

Latency-reducing techniques. Shandian is compatible with existing latency-reduction techniques with notable examples of caching and CDNs. Both caching and CDNs use a URL as the key to store a Web object. To preserve the use of caching and CDNs, we need to preserve the integrity of both the Web object itself and its corresponding URL. We leave images and other media unmodified because they do not block HTML parsing, and we make the design decisions to migrate unmodified CSS and JavaScript in the post-load state. All the resources that are typically cached or served from CDNs are kept unmodified, meaning that all the caching and CDN abilities are preserved.

Privacy. We consider the additional information that users have to sacrifice in order to use Shandian. Even without Shandian, websites already have access to user information revealed as part of the page load process (e.g., access patterns, user locations), and Shandian does not result in the exposure of additional user information. This is because: (i) website information stored in browsers in the form of cookies or localStorage comes from the website itself; (ii) current browsers expose geolocation to websites upon receiving consent from users; (iii) websites have already had access to the browser information (using JavaScript). To sum up, the client-side state either comes from the website or is already exposed to the website in the absence of Shandian, and thus users do not have to expose any additional information to servers in order to use Shandian.

Our design is compatible with HTTPS if it is deployed on a reverse proxy that has terminated SSL, but requires additional trust when deployed as a globally distributed proxy. Similar to Amazon Silk and Opera mini, we would need to trust the proxy. The connection between the Web server and the proxy and the connection between the proxy and the client use two separate HTTPS connections. To handle SSL certificates, we need to route the requests to the proxy so that the proxy can fetch Web pages on behalf of the client.

Security techniques backed by same-origin policies. Same-origin policy (SOP) is used to protect third-party scripts from accessing first-party assets such as cookies. Our design is compatible with SOP when third-party scripts are embedded using an `iframe`, because frames and parent document are isolated from each other. When third-party scripts are embedded using a `script` tag, they are given full permissions to access the first-party assets, in which case our design respects SOP.

Scaling the proxy servers. `Shandian` adds computation costs to proxy servers, making it hard to scale. We discuss the scalability issues of `Shandian` in three adoption scenarios: by browsers, by third-party proxy vendors, and by websites respectively. If browsers or third-party proxy vendors were to adopt `Shandian`, they can rent private cloud instances (e.g., Amazon EC2) to users, which is similar to the scenarios of Opera Mini and Amazon Silk.

If websites were to adopt `Shandian`, additional work has to be done to increase scalability. A possible approach is to exploit the similarities of the Web pages within a website. For example, when the same Web page is sent to different users, most portions of the page are the same except for personalized data. The server side of `Shandian` can cache intermediate representations that are generated from loading one Web page for one user. These intermediate representations, if used smartly, can reduce the computation of loading the same page for a different user, or for loading a different page (if there are similarities across pages). The technical details of this extension are outside of the scope of this paper.

Using a cloud-based proxy for compression. `Shandian` is orthogonal to existing cloud-based proxy approaches that do not restructure the page load process. This means that even if a proxy is already placed near the server for `Shandian`, another proxy can be placed near the client for other purposes (e.g., Android Chrome Beta [1] for data compression, SPDY proxies for rewriting connections between the proxy and the device). However, approaches that restructure the page load process at clients (e.g., Opera mini [26] and Amazon Silk [3]) cannot be used together with `Shandian`.

Extending the definition of PLT. Currently, `Shandian` is designed for improving page load times defined by the W3C `load` event. But it would be trivial to extend `Shandian` to improve any definition of page load times. The key is to capture the state of event listeners and the progress of HTML parsing for a given definition of page load time. The flexibility of PLT definitions is important because reports have shown that user-perceived page load times matter more than when the `load` event is fired [7].

7 Related Work

Cloud browsers for mobile devices. The closest related work is cloud browsers for mobile devices. Opera mini [26] and Amazon Silk [3] only handle display in client-side browsers. Therefore, evaluating JavaScript depends on the network which is demonstrated to be both slow and unreliable in mobile settings [30]. Different from these browsers, we provide a fully functioning client-side browser that reduces latencies. Mobile Chrome [1] compresses Web pages through a proxy server to reduce network traffic. Our work is orthogonal to Mobile Chrome.

Mitigating page load dependencies. To mitigate the impact of page load dependencies, SPDY server push, Klotski [10], and techniques developed by Instart Logic [20] provide means to prioritize Web contents at the object level on front-end servers, proxies, and browsers respectively. These solutions require knowledge of dependencies between Web objects within a page beforehand to build a prioritization plan. `Shandian` prioritizes Web contents at a finer granularity and does not require the system to obtain any knowledge of the Web pages beforehand. Best practices for Web authoring also aim at mitigating page load dependencies [31]. For example, a common advice is to place CSS at the beginning of a Web page and to place JavaScript at the end of a Web page. But, such advice is hard to execute since the construction of many Web pages depends on using JavaScript libraries such as jQuery [21] and D3.js [14], which need to appear above where they are used in a page. `Shandian` is the first that automatically enforces this best practice.

Improving computation. Much work has been done to improve page load computations, with a focus on exploiting parallelism. Meyerovich et al. proposed a parallel architecture for computing Web page layout by parallelizing CSS evaluations [25]. The Adrenaline browser exploits parallelism by splitting up a Web page into many pieces and processing each piece in parallel [24]. The ZOOMM browser further parallelizes the browser engine by preloading and preprocessing objects and by speeding up computation in sub-activities [12]. Due to the dependencies that are intrinsic in browsers, the level of parallelism is largely limited. `Shandian` removes dependencies for initial page loads on the client and thus provides opportunities for more parallelism. Besides increasing parallelism, other efforts focus on adding architectural support. Zhu et al. [41] specialized the processors for fast DOM tree and CSS access. Choi et al. [13] proposed a hybrid-DOM to efficiently access the DOM nodes. These approaches are orthogonal to Shandian.

Improving network. There are several efforts that improve page load latencies at the networking level. This includes speculations inside browsers (e.g., TCP preconnect [19], DNS pre-resolution [22]), using new protocols (e.g., SPDY [32], QUIC [29]), and improving TCP for Web traffic (e.g., TCP fast open [28], proportional rate reduction [16], and Tail loss probe [15, 17]). While being effective in reducing latencies for a single Web object, these techniques have a limited impact in reducing page load times; these techniques reduce the network costs but do not reduce the amount of HTTP requests for the initial page load nor do they eliminate the inefficiencies associated with dependencies. Instead, we first identify page load dependencies as the primary bottleneck for PLT, and then propose Shandian to mitigate the dependencies.

8 Summary

In this paper, we presented Shandian that improves PLT by simplifying the client-side page load process through an architecture that splits the page load process between a proxy server and the client. By performing preprocessing in the proxy server with more compute power, Shandian largely reduces the inefficiencies of page loads on the client. Shandian is fast for displaying Web pages, ensures that users are able to continue interacting with the page, and is compatible with caching, CDNs, and security features that enforce same-origin policies. Our evaluations show that Shandian reduces PLTs by more than half on a variety of mobile settings with varied RTT, bandwidth, CPU power, and memory size.

Acknowledgements

We thank our shepherd, Vyas Sekar, and the anonymous reviewers for their feedback. This work was supported by the National Science Foundation under grants CNS-1318396, CNS-1420703, and CNS-1518702.

References

[1] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Googles data compression proxy for the mobile web. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[2] Top sites in United States. http://www.alexa.com/topsites/countries/US.

[3] Amazon silk browser. http://amazonsilk.wordpress.com/.

[4] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into Web server design. In *Computer Networks Volume 33, Issue 1-6*, 2000.

[5] Blink: Chrome's Rendering Engine. http://www.chromium.org/blink.

[6] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the eye of the beholder: meeting users' requirements for Internet quality of service. In *Proc. of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), 2000.*

[7] J. Brutlag. Above the fold time: Measuring web page performance visually, Mar. 2011. http://en.oreilly.com/velocity-mar2011/public/schedule/detail/18692.

[8] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive Record/Replay for Web Application Debugging. In *Proc. of the ACM UIST, 2013.*

[9] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011.*

[10] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[11] M. Carbone and L. Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, Mar. 2010.

[12] C. Cascaval, S. Fowler, P. M. Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar. ZOOMM: A Parallel Web Browser Engine for Multicore Mobile Devices. In *Proc. of the ACM PPoPP, 2013.*

[13] R. H. Choi and Y. Choi. Designing a high-performance mobile cloud web browser. In *Proc. of the International World Wide Web Conference (WWW)*, 2014.

[14] D3.js. http://d3js.org/.

[15] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. Tail Loss Probe (TLP): An algorithm for fast recovery of tail losses, Feb. 2013.

http://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01.

[16] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi. Proportional rate reduction for TCP. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC)*, 2011.

[17] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: the virtue of gentle aggression. In *Proc. of the ACM SIGCOMM*, 2013.

[18] Google. Content module. http://www.chromium.org/developers/content-module.

[19] I. Grigorik. Chrome networking: DNS prefetch & TCP preconnect, June 2012. http://www.igvita.com/2012/06/04/chrome-networking-dns-prefetch-and-tcp-preconnect/.

[20] Instart logic. https://www.instartlogic.com/.

[21] jquery. https://www.jquery.com/.

[22] E. Lawrence. Internet Explorer 9 network performance improvements, Mar. 2011. http://blogs.msdn.com/b/ie/archive/2011/03/17/internet-explorer-9-network-performance-improvements.aspx.

[23] J. Lo, E. Wohlstadter, and A. Mesbah. Imagen: Runtime Migration of Browser Sessions for JavaScript Web Applications. In *Proc. of the International World Wide Web Conference (WWW), 2013*.

[24] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A Case for Parallelizing Web Pages. In *Proc. of HotPar, 2012*.

[25] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proc. of the international conference on World Wide Web (WWW), 2010*.

[26] Opera mini browser. http://www.opera.com/mobile/.

[27] Shopzilla: faster page load time = 12% revenue increase. http://www.strangeloopnetworks.com/resources/infographics/web-performance-and-ecommerce/shopzilla-faster-pages-12-revenue-increase/.

[28] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2011.

[29] J. Roskind. QUIC, a multiplexed stream transport over UDP. http://www.chromium.org/quic.

[30] A. Sivakumar, V. Gopalakrishnan, S. Lee, S. Rao, S. Sen, and O. Spatscheck. Cloud is not a silver bullet: A Case Study of Cloud-based Mobile Browsing. In *Proc. of HotMobile, 2014*.

[31] S. Souders. *High Performance Web Sites*. O'Reilly Media, 2007.

[32] Spdy. http://dev.chromium.org/spdy.

[33] SPDY whitepaper. http://www.chromium.org/spdy/spdy-whitepaper.

[34] Speed index. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

[35] Apache module for rewriting web pages to reduce latency and bandwidth. http://www.modpagespeed.com/.

[36] V8: Chrome's JavaScript Engine. https://developers.google.com/v8/.

[37] Cascading Style Sheets level 2 revision 1 (CSS 2.1) specification, June 2011. http://www.w3.org/TR/CSS21/.

[38] Document Object Model (DOM) Level 3 Events specification, Sept. 2014. http://www.w3.org/TR/DOM-Level-3-Events/.

[39] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2013*.

[40] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2014.

[41] Y. Zhu and V. J. Reddi. WebCore: Architectural Support for Mobile Web Browsing. In *Proc. of the 41st International Symposium on Computer Architecture (ISCA)*, 2014.

Polaris: Faster Page Loads Using Fine-grained Dependency Tracking

Ravi Netravali[*], Ameesh Goyal[*], James Mickens[†], Hari Balakrishnan[*]
[*]MIT CSAIL [†]Harvard University

Abstract

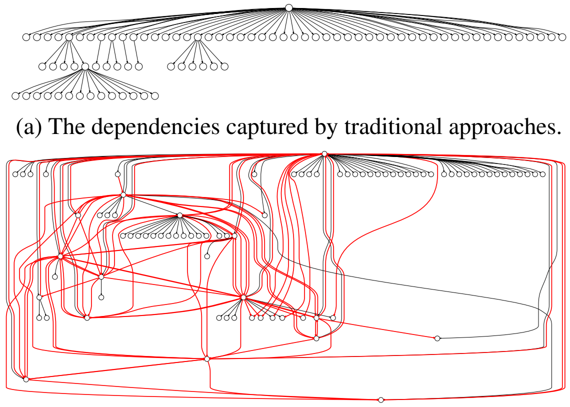
To load a web page, a browser must fetch and evaluate objects like HTML files and JavaScript source code. Evaluating an object can result in additional objects being fetched and evaluated. Thus, loading a web page requires a browser to resolve a *dependency graph*; this partial ordering constrains the sequence in which a browser can process individual objects. Unfortunately, many edges in a page's dependency graph are unobservable by today's browsers. To avoid violating these hidden dependencies, browsers make conservative assumptions about which objects to process next, leaving the network and CPU underutilized.

We provide two contributions. First, using a new measurement platform called Scout that tracks fine-grained data flows across the JavaScript heap and the DOM, we show that prior, coarse-grained dependency analyzers miss crucial edges: across a test corpus of 200 pages, prior approaches miss 30% of edges at the median, and 118% at the 95th percentile. Second, we quantify the benefits of exposing these new edges to web browsers. We introduce Polaris, a dynamic client-side scheduler that is written in JavaScript and runs on unmodified browsers; using a fully automatic compiler, servers can translate normal pages into ones that load themselves with Polaris. Polaris uses *fine-grained dependency graphs* to dynamically determine which objects to load, and when. Since Polaris' graphs have no missing edges, Polaris can aggressively fetch objects in a way that minimizes network round trips. Experiments in a variety of network conditions show that Polaris decreases page load times by 34% at the median, and 59% at the 95th percentile.

1 Introduction

Users demand that web pages load quickly. Extra delays of just a few milliseconds can result in users abandoning a page early; such early abandonment leads to millions of dollars in lost revenue for page owners [5, 6, 10]. A page's load time also influences how the page is ranked by search engines—faster pages receive higher ranks [12]. Thus, a variety of research projects [17, 23, 33, 34] and commercial systems [1, 21, 22, 31] have tried to reduce page load times.

To load a page, a browser must resolve the page's *dependency graph* [8, 18, 37]. The dependency graph captures "load-before" relationships between a page's HTML, CSS, JavaScript, and image objects. For example, a browser must parse the HTML `<script>` tag for a JavaScript file before that file can be fetched. Similarly, the browser must execute the JavaScript code in that file to reveal which images should be dynamically fetched via `XMLHttpRequests`. The overall load time for a page is the time that the browser needs to resolve the page's dependency graph, fetch the associated objects, and evaluate those objects (e.g., by rendering images or executing JavaScript files). Thus, fast page loads require efficient dependency resolution.


(a) The dependencies captured by traditional approaches.

(b) The dependencies captured by Scout, which tracks fine-grained data flows. New edges are shown in red.

Figure 1: Dependency graphs for weather.com.

Unfortunately, a page's dependency graph is only partially revealed to a browser. As a result, browsers must use conservative algorithms to fetch and evaluate objects, to ensure that hidden load-before relationships are not violated. For example, consider the following snippet of HTML:

```
<script src=``http://x.com/first.js''/>
<script src=``http://y.com/second.js''/>
<img src=``http://z.com/photo.jpg''/>
```

When a browser parses this HTML and discovers the first `<script>` tag, the browser must halt the parsing and rendering of the page, since the evaluation of `first.js` may alter the downstream HTML [19]. Thus, the browser must *synchronously* fetch and evaluate `first.js`; this is true even if `first.js` does not modify the downstream HTML or define JavaScript state required by `second.js`. Synchronously loading JavaScript files guarantees correctness, but this approach is often too cautious. For example, if `first.js` and `second.js` do not modify mutually observable state, the browser should be free to download and evaluate the files in whatever order maximizes the utilization of the

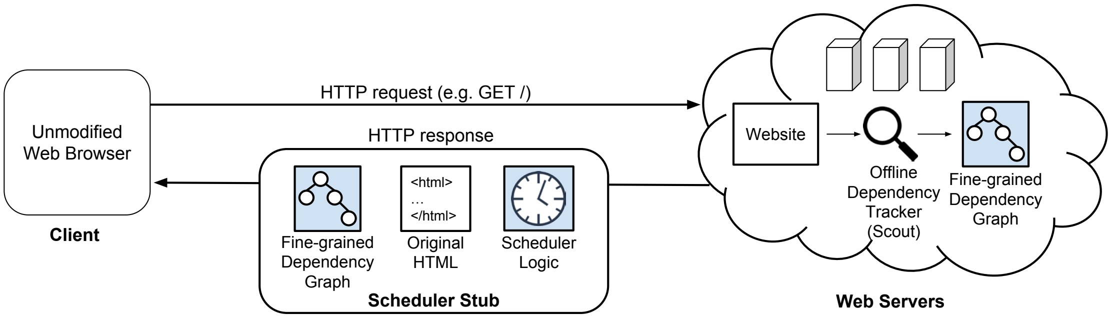

HTTP request (e.g. GET /)
Unmodified Web Browser
HTTP response
Client
<html> … </html>
Fine-grained Dependency Graph
Original HTML
Scheduler Logic
Scheduler Stub
Website
Offline Dependency Tracker (Scout)
Fine-grained Dependency Graph
Web Servers

Figure 2: With Polaris, clients request web pages using standard HTTP requests. Servers return a page's HTML, as well as the Polaris scheduler (written in JavaScript) and the page's fine-grained dependency graph (generated offline by Scout). Polaris then determines the best order to fetch the external objects that are referenced by the HTML.

network and the CPU. However, pages do not expose such fine-grained dependency information to browsers. This forces browsers to make conservative assumptions about safe load orders by using coarse-grained relationships between HTML tags to guide object retrieval. As a result, pages load more slowly than necessary.

This paper makes two contributions. First, we introduce a new measurement infrastructure called Scout that automatically tracks *fine-grained data dependencies* in web pages. By rewriting JavaScript code and HTML files, Scout instruments web pages to track precise data flows between and within the JavaScript heap and the browser's internal HTML and CSS state. For example, Scout can track read/write dependencies for an individual JavaScript variable that is accessed by multiple JavaScript files. The resulting dependency graphs are more accurate than those of prior frameworks. As shown in Figure 1, our graphs also have dramatically different structures than those of previous approaches. In particular, for 81% of the 200 real-world pages that we examined, our new graphs have *different critical paths* than those of graphs from prior work (§3.5). The critical path defines the set of object evaluations which, if delayed, will always delay the end-to-end load time for a page. Thus, the fact that our new graphs look different is not just an academic observation: our graphs imply a faster way to load web pages.

Our second contribution is Polaris, a dynamic client-side scheduler which uses Scout's fine-grained dependency graphs to reduce page load times. Figure 2 provides an overview of how Polaris works. When a user makes a request for a Polaris-enabled page, the server returns a scheduler stub instead of the page's original HTML. The scheduler stub includes the Polaris JavaScript library, the page's fine-grained dependency graph (as generated by Scout), and the original HTML. The Polaris library uses the Scout graph, as well as dynamic observations about network conditions, to load objects in an order that reduces page load time.

As shown in Figure 1, our fine-grained data tracking *adds* new constraints to standard dependency graphs. However, and perhaps counterintuitively, the Polaris scheduler has *more* opportunities to reduce page load times. The reason is that, since Polaris has a priori knowledge of the true data dependencies in a page, Polaris can aggressively fetch and evaluate objects "out-of-order" with respect to lexical constraints between HTML tags. In contrast, prior scheduling frameworks lack knowledge of many dependencies, and are forced to make conservative assumptions that are derived from the lexical HTML relationships (§2.2). Those conservative assumptions guarantee the correctness of an assembled page in the face of hidden dependencies, but they often leave a browser's CPU and network connections underutilized. By using fine-grained dependency graphs, Polaris can ensure both correctness *and* high utilization of processors and network connections.

Because Polaris' scheduler is implemented in JavaScript, Polaris can reduce page load times on *unmodified commodity browsers*; this contrasts with load optimizers like Klotski [8], Amazon Silk [3], and Opera Mini [30], which require modified browsers to interact with a server-side component. Polaris is also complementary to previous load optimizers that use data compression (§6) or multiplex several HTTP requests atop a single TCP connection (§5.4).

We evaluated Polaris using 200 popular web pages and a variety of network conditions, with latencies ranging from 25 ms to 500 ms, and bandwidths ranging from 1 Mbit/s to 25 Mbits/s. Polaris reduced page load times by 34% at the median, and 59% for the 95th percentile sites.

2 BACKGROUND

In a conventional page load, the browser first downloads the page's top-level HTML. For now, we assume that the HTML does not reference any JavaScript, CSS, or multimedia files. As the browser parses the HTML tags, it generates a data structure called the Document Object Model (DOM) tree. Each HTML tag has a corresponding node in the DOM tree; the overall structure of the DOM tree mirrors the hierarchical tag structure of the HTML. Once the HTML parse is finished and the DOM tree is complete, the browser constructs a render tree, which only contains the DOM nodes to be displayed. For example, a `<text>` node is renderable, but a `<head>` node is not. Each node in the render tree is tagged with visual attributes like background color, but render nodes do not possess on-screen positions or sizes. To calculate those geometric properties, the browser traverses the render tree and produces a layout tree, which determines the spatial location of all renderable tags. Finally, the browser traverses the layout tree and updates (or "paints") the screen. Modern browsers try to pipeline the construction of the various trees, in order to progressively display a page.

2.1 Loading More Complicated Pages

JavaScript: Using `<script>` tags, HTML can include JavaScript code. A script tag blocks the HTML parser, halting the construction of the DOM tree and the derivative data structures. Script tags block HTML parsing because JavaScript can use interfaces like `document.write()` to dynamically change the HTML after a `<script>` tag; thus, when the HTML parser encounters a `<script>` tag, the parser cannot know what the post-`<script>` HTML will look like until the JavaScript code in the tag has executed. As a result, script tags inject synchronous JavaScript execution delays into a page load. If a script tag does not contain inline source code, the browser also incurs network latencies to download the JavaScript code.

To reduce these synchronous latencies, modern browsers allow developers to mark a `<script>` tag with the `async` or `defer` attribute. An `async` script is downloaded in parallel with the HTML parse, but once it is downloaded, it will execute synchronously, in a parse-blocking manner. A `defer` script is only downloaded and executed once HTML parsing is complete.

By default, a `<script>` tag is neither `async` nor `defer`. Such scripts represent 98.3% of all JavaScript files in our test corpus of 200 popular sites (§3.5). When the HTML parser in a modern browser encounters a synchronous `<script>` tag, the parser enters *speculation mode.* The parser initiates the download of the JavaScript file, and as that download completes in the background, the parser continues to process the HTML after the script tag, fetching the associated objects and updating a speculative version of the DOM. The browser discards the speculative DOM if it is invalidated by the execution of the upstream JavaScript code. We demonstrate in Section 5 that speculative parsing is limited in its ability to resolve deep dependency chains consisting of multiple JavaScript files.

Object Type	Median	95th Percentile
HTML	11.8%	26.2%
JavaScript	22.9%	43.0%
CSS	3.7%	16.7%
Images	44.9%	77.4%
Fonts	0.0%	7.8%
JSON	0.4%	5.0%
Other	0.0%	7.8%

Table 1: Per-page object distributions for 200 popular sites.

CSS: A page may use CSS to define the visual presentation of HTML tags. The browser represents those stylings using a CSS Object Model (CSSOM) tree. The root of the CSSOM tree contains the general styling rules that apply to all HTML tags. Different paths down the tree apply additional rules to particular types of nodes, resulting in the "cascading" aspect of cascading style sheets.

A browser defines a default set of CSS rules known as the user agent styles. A web page provides additional rules by incorporating CSS `<link>` tags. To create the render tree, the browser uses the DOM tree to enumerate a page's visible HTML tags, and the CSSOM tree to determine what those visible tags should look like.

CSS tags do not block HTML parsing, but they do block rendering, layout, and painting. The reason is that unstyled pages are visually unattractive and potentially non-interactive, so style computations should be handled promptly. Best practices encourage developers to place CSS tags at the top of pages, to ensure that the CSSOM tree is built quickly. Since JavaScript code can query the CSS properties of DOM nodes, the browser halts JavaScript execution while CSS is being processed; doing so avoids race conditions on CSS state.

Images: Browsers do not load `<img>` tags synchronously. Thus, a page can be completely rendered and laid out (and partially painted) even if there are outstanding image requests. However, browsers are still motivated to load images as quickly as possible, since users do not like pages with missing images.

Other media files: Besides images, a page can include various types of video and audio files. However, in this paper, we focus on the loading of HTML, JavaScript, CSS, and image files, which are the most common types of web objects (see Table 1). Optimizing the loading process for rich multimedia files requires complex, media-specific techniques (e.g., [11, 15]).

2.2 The Pitfalls of Lexical Dependencies

As described above, the traditional approach for loading a page is constrained by uncertainty. For example:

- A script tag *might* read CSS style properties from the DOM tree, so CSS evaluation must block JavaScript execution.
- A script tag *might* change downstream HTML, so when the browser encounters a script tag, either HTML parsing must block (increasing page load time), or HTML parsing must transfer to a speculative thread (a thread which, if aborted, will have wasted network and computational resources).
- In the example from Section 1, two script tags that are lexically adjacent *might* exhibit a write/read dependency on JavaScript state. Thus, current browsers must execute the script tags serially, in lexical order, even if a different order (or parallel execution) would be more efficient.

These inefficiencies arise because HTML expresses a strict tag ordering that is based on *lexical dependencies* between tags. In reality, a page's true dependency graph is a partial ordering in which edges represent *true semantic dependencies* like write/read dependencies on JavaScript state. Since HTML does not express all of the true semantic dependencies, the browser is forced to pessimistically guess those dependencies, or use optimistic speculation that may waste resources.

In Section 3, we enumerate the kinds of true semantic dependencies that pages can have, and introduce a new framework to extract them. In Section 4, we describe how developers can expose true dependencies to the browser, allowing the browser to load pages faster.

3 Dependency Tracking

In a traditional dependency graph [8, 13, 18, 25, 26], a vertex represents an object like an image or a JavaScript file. An edge represents a load-before relationship that is the side-effect of parsing activity. For example, if a page incorporates an image via an `<img>` tag, the image's parent in the dependency graph will be the HTML file which contains the tag; if an image is fetched via an `XMLHttpRequest`, the image's parent will be the associated JavaScript file.

By emphasizing fetch initiation contexts, i.e., the file whose parsing causes an object to be downloaded, traditional dependency graphs mimic the lexical restrictions that constrain real browsers (§2). However, fetch initiation contexts obscure the fine-grained data flows that truly govern the order in which a page's objects must be assembled. In this section, we provide a taxonomy for those fine-grained dependencies, and describe a new measurement framework called Scout that captures those dependencies. The resulting dependency graphs have more edges than traditional graphs (because finer-grained dependencies are included). However, as we show in Section 5, fine-grained dependency graphs permit *more* aggressive load schedules, because browsers are no longer shackled by conservative assumptions about where hidden dependencies might exist.

3.1 Page State

Objects in a web page interact with each other via two kinds of state. The *JavaScript heap* contains the code and the data that are managed by the JavaScript runtime. This runtime interacts with the rest of the browser through the DOM interface. The DOM interface reflects internal, C++ browser state into the JavaScript runtime. However, the reflected JavaScript objects do not directly expose the rendering and layout trees. Instead, the DOM interface exposes an extended version of the DOM tree in which each node also has properties for style information and physical geometry (§2). By reading and writing this *DOM state*, JavaScript code interacts with the browser's rendering, layout, and painting mechanisms. The DOM interface also allows JavaScript code to dynamically fetch new web objects, either indirectly, by inserting new HTML tags into the DOM tree, or directly, using `XMLHttpRequests` or `WebSockets`.

3.2 Dependency Types

We are interested in capturing three types of data flows that involve the JavaScript heap and the DOM state belonging to HTML and CSS.

Write/read dependencies arise when one object produces state that another object consumes. For example, `a.js` might create a global variable in the JavaScript heap; later, `b.js` might read the variable. When optimizing the load order of the two scripts, we cannot evaluate `b.js` before `a.js` (although it is safe to *fetch* `b.js` before `a.js`).

Read/write dependencies occur when one object must read a piece of state before the value is updated by another object. Such dependencies often arise when JavaScript code must read a DOM value before the value is changed by the HTML parser or another JavaScript file. For example, suppose that the HTML parser encounters a JavaScript tag that lacks the `async` or `defer` attributes. The browser must synchronously execute the JavaScript file. Suppose that the JavaScript code reads the number of DOM nodes that are currently in the DOM tree. The DOM query examines a snapshot of the DOM tree at a particular moment in time; as explained in Section 2, a browser progressively updates the DOM tree as HTML is parsed. Thus, any reordering of object evaluations must ensure value equivalence for DOM queries—regardless of when a JavaScript file is executed, its DOM queries must return the same results. This guarantees deterministic JavaScript execution semantics [24] despite out-of-order evaluation.

Write/write dependencies arise when two objects update the same piece of state, and we must preserve the relative ordering of the writes. For example, CSS files update DOM state, changing the rules which govern a page's visual presentation. The CSS specification states that, if two files update the same rule, the last writer wins. Thus, CSS files which touch the same rule must be evaluated in their original lexical ordering in the HTML. However, the evaluation of the CSS files can be arbitrarily reordered with respect to the execution of JavaScript code that does not access DOM state.

Output devices are often involved in write/write dependencies. As described in the previous paragraph, CSS rules create a write/write dependency on a machine's display device. Write/write dependencies can also arise for local storage and the network. For example, the `localStorage` API exposes persistent storage to JavaScript using a key/value interface. If we shuffle the order in which a page evaluates JavaScript objects, we must ensure that the final value for each `localStorage` key is the same value that would result from the original execution order of the JavaScript files.

Traditional dependencies based on HTML tag constraints can often be eliminated if finer-grained dependencies are known. For example, once we know the DOM dependencies and JavaScript heap dependencies for a `<script>` tag, the time at which the script can be evaluated is completely decoupled from the position of the `<script>` tag in the HTML—we merely have to ensure that we evaluate the script after its fine-grained dependencies are satisfied. Similarly, we can parse and render a piece of HTML at any time, as long as we ensure that we have blocked the evaluation of downstream objects in the dependency graph.

Images do need to be placed in specific locations in the DOM tree. However, browsers already allow images to be fetched and inserted asynchronously. So, images can be fetched in arbitrary orders, regardless of the state of the DOM tree, but their insertion is dependent on the creation of the associated DOM elements. We model this using write/write dependencies on DOM elements: the HTML parser must write an initially empty `<img>` DOM node, and then the network stack must insert the fetched image bitmap into that node.

3.3 Capturing Dependencies with Scout

To capture the fine-grained dependencies in a real web page, we first record the content of the page using Mahimahi [28]. Next, we use a new tool called Scout to rewrite each JavaScript and HTML file in the page, adding instrumentation to log fine-grained data flows across the JavaScript heap and the DOM. Scout then loads the instrumented page in a regular browser. As the page loads, it emits a dependency log to a Scout analysis server; the server uses the log to generate the fine-grained dependency graph for the page.

Tracking JavaScript heap dependencies: To track dependencies in which both actors are JavaScript code, Scout leverages JavaScript proxy objects [27]. A proxy is a transparent wrapper for an underlying object, allowing custom event handlers to fire whenever external code tries to read or write the properties of the underlying object.

In JavaScript, the global namespace is explicitly nameable via the `window` object; for example, the global variable `x` is also reachable via the name `window.x`. Scout's JavaScript rewriter transforms unadorned global names like `x` to fully qualified names like `window.x`. Also, for each JavaScript file (whether inline or externally fetched), Scout wraps the file's code in a closure which defines a local alias for the `window` variable. The aliasing closures, in combination with rewritten code using fully qualified global names, forces all accesses to the global namespace to go through Scout's `window` proxy. Using that proxy, Scout logs all reads and writes to global variables.

Scout's `window` proxy also performs recursive proxying for non-primitive global values. For example, reading a global object variable `window.x` returns a logging proxy for that object. In turn, reading a non-primitive value `y` on that proxy would return a proxy for `y`. By using recursive proxying and wrapping calls to `new` in proxy generation code, Scout can log any JavaScript-issued read or write to JavaScript state. Each read or write target is logged using a fully qualified path to the `window` object, e.g., `window.x.y.z`. Log entries also record the JavaScript file that issued the operation.

Scout's proxy generation code tags each underlying object with a unique, non-enumerable integer id. The proxy code also stores a mapping between ids and the corresponding proxies. When a proxy for a particular object is requested, Scout checks whether the object already has an id. If it does, Scout returns the preexisting proxy for that object, creating proxy-level reference equalities which mirror those of the underlying objects.

Some objects lack a fully-qualified path to `window`. For example, a function may allocate a heap object and return that object to another function, such that neither function assigns the object to a variable that is recursively reachable from `window`. In these cases, Scout logs the identity of the object using the unique object id.

Tracking DOM dependencies: JavaScript code interacts with the DOM tree through the `window.document` object. For example, to find the DOM node with a particular id, JavaScript calls document.getElementById(id). The DOM nodes that are returned by document provide additional interfaces for adding and removing DOM nodes, as well as changing the CSS properties of those nodes.

To track dependencies involving JavaScript code and DOM state, Scout's recursive proxy for window.document automatically creates proxies for all DOM nodes that are returned to JavaScript code. For example, the DomNode returned by document.getElementById(id) is wrapped in a proxy which logs reads and writes to the object via interfaces like DomNode.height.

Developers do not assign ids to most DOM nodes. Thus, Scout's logs identify DOM nodes by their paths in the DOM tree. For example, the DOM path <1,5,2> represents the DOM node that is discovered by examining the first child of the HTML tag, the fifth child of that tag, and then the second child of that tag.

A write to a single DOM path may trigger cascading updates to other paths; Scout must track all of these updates. For example, inserting a new node at a particular DOM path may shift the subtrees of its new DOM siblings to the right in the DOM tree. In this case, Scout must log writes to the rightward DOM paths, as well as to the newly inserted node. Similar bookkeeping is necessary when DOM nodes are deleted or moved to different locations.

The DOM tree can also be modified by the evaluation of CSS objects that change node styles. Scout models each CSS tag as reading all of the DOM nodes that are above it in the HTML, and then writing all of those DOM nodes with new style information. To capture the set of affected DOM nodes, Scout's HTML rewriter prepends each CSS tag with an inline JavaScript tag that logs the current state of the DOM tree (i.e., all of the live DOM paths) and then deletes itself from the DOM tree.

In Scout logs, we represent DOM operations using the window.$$dom pseudovariable. For example, the identifier window.$$dom.1 represents the first child of the topmost <html> node. We also use the window.$$xhr pseudovariable to track network reads and writes via XMLHttpRequests. These pseudovariables allow us to use a single analysis engine to process all dependency types.

Missing Dependencies: To generate a page's dependency graph, Scout loads an instrumented version of the page on a server-side browser, and collects the resulting dependency information. Later, when Polaris loads the page on a client-side browser (§4), Polaris assumes that Scout's dependency graph is an accurate representation of the dependencies in the page. This might not be true if the page's JavaScript code exhibits nondeterministic behavior. For example, suppose that a page contains three JavaScript files called a.js, b.js, and c.js. At runtime, a.js may call Math.random(), and use the result to invoke a function in b.js or c.js (but not both). During some executions, Scout will log a dependency between a.js and b.js; during other executions, Scout will log a dependency between a.js and c.js. If there is a discrepancy between the dependency logged by Scout, and the dependency generated by the code on the client browser, then Polaris may evaluate JavaScript files in the wrong order, breaking correctness.

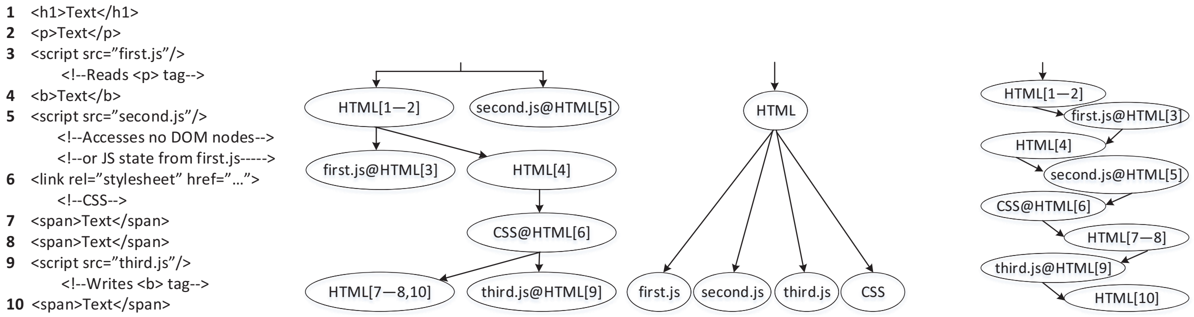
We have not observed such nondeterministic dependencies in our corpus. However, if a page does include such dependencies, Scout must create a dependency graph which contains the aggregate set of all possible dependencies. Such a graph overconstrains any particular load of the page, but guarantees that clients will load pages without errors. The sources of nondeterministic JavaScript events are well-understood [24], so Scout can use a variety of techniques to guarantee that nondeterministic dependencies are either tracked or eliminated. For example, Scout can rewrite pages so that calls to Math.random() use a deterministic seed [24], removing nondeterminism from calls to the random number generator.

For a given page, a web server may generate a different dependency graph for different clients. For example, a web server might personalize the graph in response to a user's cookie; as another example, a server might return a smaller dependency graph in response to a user agent string which indicates a mobile browser. The server-side logic must run Scout on each version of the dependency graph. We believe that this burden will be small in practice, since even customized versions of a page often share the same underlying graph structure (with different content in some of the nodes).

Implementation: To build Scout, we used Esprima [14], Estraverse [36], and Escodegen [35] to rewrite JavaScript code, and we used Beautiful Soup [32] to rewrite HTML. We loaded the instrumented pages in a commodity Firefox browser (version 40.0). Each page sent its dependency logs to a dedicated analysis server; logs were sent via an XMLHttpRequest that was triggered by the onload event.

Our implementation of Scout handles the bulk of the JavaScript language. However, our implementation does not currently support the eval(sourceCode) statement, which pages use to dynamically execute new JavaScript code. To support this statement, Scout would need to shim eval() and dynamically rewrite the sourceCode argument so that the rewritten code tracked dependencies.

Our current implementation also does not support the with(obj) statement, which places obj at the beginning of the scope chain that the JavaScript runtime



```
1  <h1>Text</h1>
2  <p>Text</p>
3  <script src="first.js"/>
       <!--Reads <p> tag-->
4  <b>Text</b>
5  <script src="second.js"/>
      <!--Accesses no DOM nodes-->
      <!--or JS state from first.js----->
6  <link rel="stylesheet" href="...">
      <!--CSS-->
7  <span>Text</span>
8  <span>Text</span>
9  <script src="third.js"/>
       <!--Writes <b> tag-->
10 <span>Text</span>
```

(a) The HTML for a simple page.

(b) The dependency graph generated by Scout.

(c) The dependency graph created by Klotski [8].

(d) The dependency graph created by WProf [37].

Figure 3: Comparing the order in which different tools declare that a simple page's objects must be evaluated. The notation HTML[i:j] refers to HTML lines i up to and including j. The notation obj@HTML[k] refers to the object whose corresponding tag is at HTML[k].

uses to resolve variable names. To support this statement, Scout merely needs to wrap the `obj` argument in code which checks whether `obj` is a proxy; if not, the wrapper would return one.

3.4 Dependency Graphs: Scout vs. Prior Tools

Figure 3(a) depicts a simple web page with two JavaScript files and one CSS file. Figures 3(b), (c), and (d) show the dependency graphs that are produced by Scout, Klotski [8], and WProf [37].

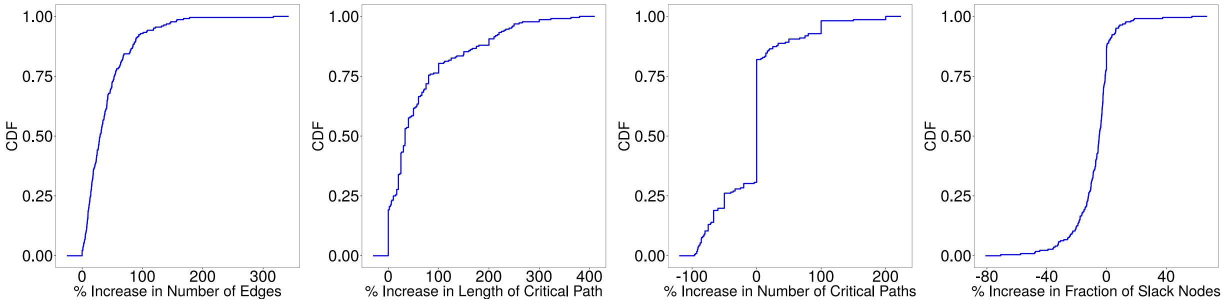
- Scout allows `second.js` and the first chunk of HTML to be evaluated in parallel, since second.js does not access DOM state or JavaScript state defined by prior JavaScript files. `first.js` does access DOM state from upstream HTML tags, but Scout allows the evaluation of `first.js` to proceed in parallel with the parsing of downstream HTML. Scout treats CSS as a read and then a write to all upstream HTML, so the CSS file must be evaluated before the evaluation of downstream HTML and downstream scripts which access DOM state.
- Klotski [8] cannot observe fine-grained data flows, so its dependency graphs are defined by lexical HTML constraints (§2). Given a dependency graph like the one shown in Figure 3(c), Klotski uses heuristics to determine which objects a server should push to the browser first. However, Klotski does not know the page's true data dependencies, so Klotski cannot guarantee that prioritized objects can actually *evaluate* ahead of schedule with respect to their evaluation times in the original page. It is only safe to evaluate an object (prioritized or not) when its ancestors in the dependency graph have been evaluated. So, Klotski's prioritized pushes can safely warm the client-side cache, but in general, it is unsafe for those pushes to synchronously trigger object evaluations.
- By instrumenting the browser, WProf observes the times at which a browser is inside the network stack or a parser for HTML, CSS, or JavaScript. Thus, WProf can track complex interactions between a browser's fetching, parsing, and evaluation mechanisms. However, this technique only allows WProf to analyze the critical path for the *lexically-defined* dependency graph. This graph does not capture true data flows, and forces conservative assumptions about evaluation order (§2.2). As shown in Figure 3(d), WProf overconstrains the order in which objects can be evaluated (although WProf may allow objects to be *fetched* out-of-lexical-order).

In summary, only Scout produces a dependency graph which captures the true constraints on the order in which objects can be evaluated. Polaris uses these fine-grained dependencies to schedule object downloads—by prioritizing objects that block the most downstream objects, Polaris reduces overall page load times (§4).

3.5 Results

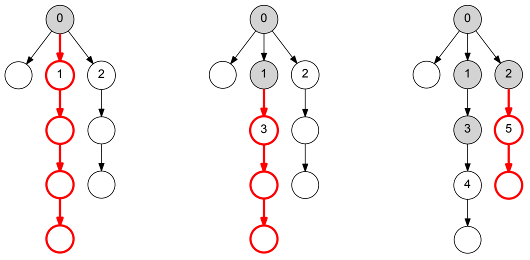
We used Mahimahi [28], an HTTP record-and-replay tool, to record the content from 200 sites in the Alexa Top 500 list [2]. The corpus spanned a variety of page categories, including news, ecommerce, and social media. The corpus also included five mobile-optimized sites. Since our Scout prototype does not support the `eval()` and `with()` statements, we selected pages which did not use those statements.

Figure 4 summarizes the differences between Scout's dependency graphs and the traditional ones that are defined by Klotski [8] and the built-in developer tools from Chrome [13], Firefox [26], and IE [25]. As shown in Figure 4(a), traditional graphs are almost always incomplete, missing many edges that can only be detected via data flow analysis. That analysis adds 29.8% additional



(a) (b) (c) (d)

Figure 4: How traditional dependency graphs change when updated with information from fine-grained data flows. The updated graphs have additional edges which belong to previously untracked dependencies. The new edges often modify a page's critical paths. Note that a slack node is a node that is not on a critical path.



(a) (b) (c)

Figure 5: An example of dynamic critical paths during the load of a simple page. Dynamic critical paths are shown in red. Numbers represent the order in which Polaris requests the objects. Shaded objects have been received and evaluated; numbered but unshaded objects have been requested, but have no responses yet. We assume that all objects are from the same origin, and that only two outstanding requests per origin are allowed.

edges at the median, and 118% more edges at the 95th percentile.

Those additional edges have a dramatic impact on the characteristics of dependency graphs. For example, **adding fine-grained dependencies alters the critical path length for 80.8% of the pages in our corpus** (Figure 4(b)). The set of objects on those paths often changes, with old objects being removed and new objects being added. Furthermore, as shown in Figure 4(d), **86.6% of pages have a smaller fraction of slack nodes when fine-grained dependencies are considered.** Slack nodes are nodes that are not on a critical path. Thus, a decrease in slack nodes means that browsers have *fewer* load schedules which result in optimal page load times.

4 Polaris: Dynamic Client-side Scheduling

Polaris is a client-side scheduler for the loading and evaluation of a page's objects. Polaris is written completely in JavaScript, allowing it to run on unmodified commodity browsers. Polaris accepts a Scout graph as input, but also uses observations about current network conditions to determine the *dynamic critical path* for a page. The dynamic critical path, i.e., the path which *currently* has the most unresolved objects, is influenced by the order and latency with which network fetches complete; importantly, the dynamic critical path may be different than the critical path in the static dependency graph.[1] Polaris prioritizes the fetching and evaluation of objects along the dynamic critical path, trying to make parallel use of the client's CPU and network, and trying to keep the client's network pipe full, given browser constraints on the maximum number of simultaneous network requests per origin.

Figure 5 shows how a page's dynamic critical path can change over time. In Figure 5(a), Polaris has evaluated object 0, and issued requests for objects 1 and 2, because those objects are the roots for the deepest unresolved paths in the dependency graph. In Figure 5(b), Polaris has received and evaluated object 1, although object 2 is still in-flight. Polaris has one available request slot, so it requests object 3, because that object is the root of the deepest unresolved path. In Figure 5(c), Polaris has received and evaluated object 3; Polaris uses the available request slot to fetch object 4. Then, object 2 is received and evaluated. The critical path changes—the deepest chain is now beneath object 2, so Polaris requests object 5 next.

To use Polaris with a specific page, a web developer runs Scout on that page to generate a dependency graph and a *Polaris scheduler stub*. The developer then configures her web server to respond to requests for that page with the scheduler stub's HTML instead of the page's

[1] This is why a dynamic client-side scheduler is better than a static client-side scheduler that ignores current network conditions and deterministically fetches objects from a server-provided URL list.

regular HTML (see Figure 2). The stub contains four components.

- The **scheduler** itself is just inline JavaScript code.
- The **Scout dependency graph** for the page is represented as a JavaScript variable inside the scheduler.
- **DNS prefetch hints** indicate to the browser that the scheduler will be contacting certain hostnames in the near future. DNS prefetch hints are expressed using `<link>` tags of type `dns-prefetch`, e.g.,

  ```
  <link rel=''dns-prefetch''
      href=''http://domain.com''>
  ```

  DNS hints allow Polaris to pre-warm the DNS cache in the same way that the browser does during speculative HTML parsing (§2.1).
- Finally, the stub contains **the page's original HTML**, which is broken into chunks as determined by Scout's fine-grained dependency resolution (see §3.3 and Figure 3). When Scout generates the HTML chunks, it deletes all `src` attributes in HTML tags, since the external objects that are referenced by those attributes will be dynamically fetched and evaluated by Polaris.

Polaris adds few additional bytes to a page's original HTML. Across our test corpus of 200 sites, the scheduler stub was 3% (36.5 KB) larger than a page's original HTML at the median.

The scheduler uses `XMLHttpRequests` to dynamically fetch objects. To evaluate a JavaScript file, the scheduler uses the built-in `eval()` function that is provided by the JavaScript engine. To evaluate HTML, CSS, and images, Polaris leverages DOM interfaces like `document.innerHTML` to dynamically update the page's state.

In the rest of this section, we discuss a few of the subtler aspects of implementing an object scheduler as a JavaScript library instead of native C++ code inside the browser.

Browser network constraints: Modern browsers limit a page to at most six outstanding requests to a given origin. Thus, Polaris may encounter situations in which the next missing object on the dynamic critical path would be the seventh outstanding request to an origin. If Polaris actually generated the request, the request would be placed at the end of the browser's internal network queue, and would be issued at a time of the browser's choosing. Polaris would lose the ability to precisely control the in-flight requests at any given moment.

To avoid this dilemma, Polaris maintains per-origin priority queues. With the exception of the top-level HTML (which is included in the scheduler stub), each object in the dependency graph belongs to exactly one queue. Inside a queue, objects that are higher in the dependency tree receive a higher priority, since those objects prevent the evaluation of more downstream objects. At any given moment, the scheduler tries to fetch objects that reside in a dynamic critical path for the page load. However, if fetching the next object along a critical path would violate a per-origin network constraint, Polaris examines its queues, and fetches the highest priority object from an origin that has available request slots.[2]

Frames: A single page may contain multiple iframes. Scout generates a scheduler stub for each one, but the browser's per-origin request cap is a page-wide limit. Thus, the schedulers in each frame must cooperate to respect the limit and prevent network requests from getting stuck inside the browser's internal network queues.

The scheduler in the top frame coordinates the schedulers in child frames. Using `postMessage()` calls, children ask the top-most parent for permission to request particular objects. The top-most parent only authorizes a fetch if per-origin request limits would not be violated.

URL matching: A page's coarse-grained dependency graph has a stable structure [8]. In other words, the edges and vertices that are defined by lexical HTML constraints change slowly over time. However, the URLs for specific vertices change more rapidly. For example, if JavaScript code dynamically generates an `XMLHttpRequest` URL, that URL may embed the current date in its query string. Across multiple page loads, the associated object for the URL will have different names, even though all of the objects will reside in the same place in the dependency graph.

To handle any discrepancies between the URLs in Scout's dependency graphs and the URLs which `XMLHttpRequests` generate on the client, Polaris uses a matching heuristic to map dynamic URLs to their equivalents in the static dependency graph. Our prototype implementation uses Mahimahi's matching heuristic [28], but Polaris is easily configured to use others [8, 9, 33].

Page-generated XHRs: When Polaris evaluates a JavaScript file, the executed code might try to fetch an object via `XMLHttpRequest`. Assuming that a page has deterministic JavaScript code (§3.3), Scout will have included the desired object in the page's dependency graph. However, during the loading of the page in a real client browser, Polaris requires control over the order in which objects are fetched. Thus, Polaris uses an

[2] Browsers allow users to modify the constraint on the maximum number of connections per origin; Polaris can be configured to respect user-programmed values.

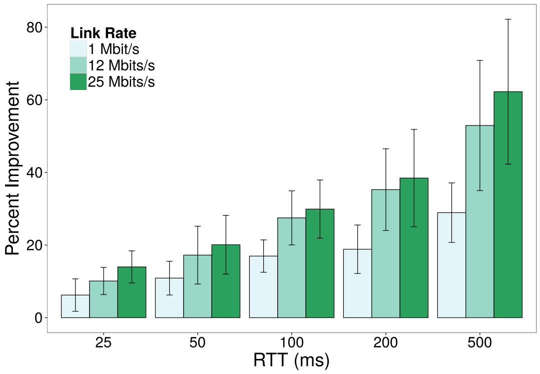


Figure 6: Polaris' average reduction in page load times, relative to baseline load times with Firefox v40.0. Each bar is the average reduction in load time across the entire 200 site corpus. Error bars span one standard deviation in each direction of the average.

		RTT		
		25 ms	100 ms	500 ms
Link Rate	1 Mbit/s	256.3 ms	883.9 ms	1857.5 ms
	12 Mbit/s	309.1 ms	1274.1 ms	2935.0 ms
	25 Mbit/s	382.5 ms	1385.3 ms	3188.3 ms

Table 2: Polaris' raw reduction in median page load times for a subset of the parameter values in Figure 6.

XMLHttpRequest shim [24] to suppress autonomous XMLHttpRequests. Polaris issues those requests using its own scheduling algorithm, and manually fires XMLHttpRequest event handlers when the associated data has arrived.

5 Results

In this section, we demonstrate that Polaris can decrease page load times across a variety of web pages and network configurations: performance improves by 34% and 59% for the median and 95th percentile sites, respectively. Polaris' benefits grow as network latencies increase, because higher RTTs increase the penalty for bad fetch schedules. Thus, Polaris is particularly valuable for clients with cellular or low-quality wired networks. However, even for networks with moderate RTTs, Polaris can often reduce load times by over 20%.

5.1 Methodology

We evaluated Polaris using the 200 site corpus that is described in Section 3.3. We used Mahimahi [28] to capture site content and later replay it using emulated network conditions. To build Polaris-enabled versions of each page, we post-processed the recorded web content, generating Polaris scheduler stubs for each site. We then compared the load times of the Polaris sites and the original versions of those sites. All experiments used Firefox v40.0. Unless otherwise specified, all experiments used cold browser caches and DNS caches.

A page's load time is normally defined with respect to JavaScript events like navigationStart and loadEventEnd. However, loadEventEnd is inaccurate for Polaris pages, since the event only indicates that the scheduler stub has been loaded; the rest of the page's objects remain to be fetched by the dynamic scheduler. So, to define the load time for a Polaris page, we first loaded the original version of the page and used tcpdump to capture the objects that were fetched between navigationStart and loadEventEnd. We then defined the load time of the Polaris page as the time needed to fetch all of those objects.

5.2 Reducing Page Load Times

Figure 6 demonstrates Polaris' ability to reduce load times. There are two major trends to note. First, for a given link rate, Polaris' benefits increase as network latency increases. For example, at a link rate of 12 Mbits/s, Polaris provides an average improvement of 10.1% for an RTT of 25 ms. However, as the RTT increases to 100 ms and 200 ms, Polaris' benefits increase to 27.5% and 35.3%, respectively. The reason is that, as network latencies grow, so do the penalties for not prioritizing the fetches of objects on the dynamic critical path. Polaris *does* prioritize the fetching of critical path objects. Furthermore, Polaris never has to wait for an object evaluation to reveal a downstream dependency—Polaris knows all of the dependencies at the beginning of the page load, so Polaris can always keep the network pipe full.

The second trend in Figure 6 is that, for a given RTT, Polaris' benefits increase as network bandwidth grows. This is because, if bandwidth is extremely low, transfer times dominate fetch costs. As bandwidth increases, latency becomes the dominant factor in download times. Since Polaris prioritizes the fetch orders for critical path objects (but does nothing to reduce those objects' bandwidth costs), Polaris' gains are most pronounced when latencies govern overall download costs.

Figure 6 describes Polaris' gains in relative terms. Table 2 depicts absolute gains, describing how many raw milliseconds of load time Polaris removes. Even on a fast network with 25 ms of latency, Polaris eliminates over 250 ms of load time. Those results are impressive, given that web developers strive to eliminate *tens of milliseconds* from their pages' load times [5, 6, 10].

The error bars in Figure 6 are large. The reason is that, for a given network bandwidth and latency, Polaris' benefits are determined by the exact structure of a page's dependency graph. To understand why, consider the three sites in Figure 7.

- The homepage for apple.com has a flat dependency graph, as shown in Figure 8. This means that, once the browser has the top-level HTML, the other objects can be fetched and evaluated in an arbitrary or-

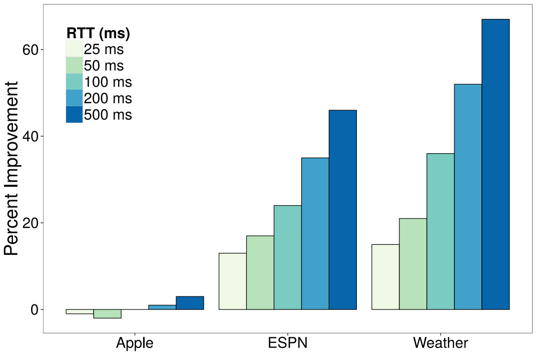


Figure 7: Polaris' average reduction in page load times, relative to baseline load times, for three sites with diverse dependency graph structures. Each experiment used a link rate of 12 Mbits/s.

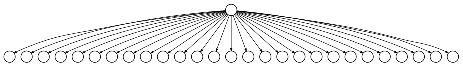


Figure 8: The dependency graph for Apple's homepage.

der; all orders will result in similar end-to-end page load times. Thus, for low RTTs, Polaris loads the apple.com homepage 1–2% *slower* than the baseline, due to computational overheads from Polaris' scheduling logic.

- In contrast, the ESPN homepage has a dependency path of length 5, and several paths of length 4. Also, 48% of the page's content is loaded from only two origins (a.espncdn.com and a1.espncdn.com), magnifying the importance of optimally scheduling the six outstanding requests for each origin (§4). In ESPN's dependency graph, many of the long paths consist of JavaScript files. However, the standard Firefox scheduler has no way of knowing this. So, when Firefox loads the standard version of the page, it initially requests a small number of JavaScript objects, and then fills the rest of its internal request queue with 32 image requests. As a result, when a JavaScript file evaluates and generates a request for another JavaScript file on the critical path, the request is often stuck behind image requests in the browser's internal network queue. In contrast, Polaris has a priori knowledge of which JavaScript files belong to deep dependency chains. Thus, Polaris prioritizes the fetching of those objects, using its knowledge of per-origin request caps to ensure that the fetches for critical path objects are never blocked.
- As shown in Figure 1(b), the weather.com homepage is even more complicated than that of ESPN. Deep, complex dependency graphs present Polaris with the most opportunities to provide gains. Thus, of the three sites in Figure 7, weather.com enjoys the largest reductions in load time.

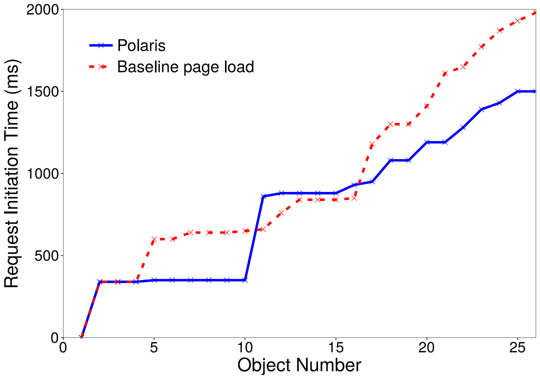


Figure 9: Request initiation times for the regular and Polaris-enabled versions of StackOverflow. These results used a 12 Mbits/s link with an RTT of 100 ms.

Figure 9 depicts the order in which requests issue for the normal version of the StackOverflow site, and the Polaris version. In general, Polaris issues requests earlier; by prioritizing the fetches of objects on the dynamic critical path, Polaris minimizes the overall fetch time needed to gather *all* objects. However, as shown in Figure 9, Polaris briefly falls behind the default browser scheduler after fetching the tenth object. The reason is that, in our current Polaris implementation, HTML is rendered in large chunks. While that HTML is being rendered, Polaris cannot issue new HTML requests, because executing Polaris' JavaScript-level scheduler would block rendering (§2.1). In contrast, a native browser scheduler can issue new requests in parallel with HTML rendering. Thus, the default Firefox scheduler has a lower time-to-first paint than Polaris, and Polaris falls behind the default scheduler after the tenth object fetch. However, after Polaris renders the bulk of the HTML, Polaris quickly regains its lead and never relinquishes it. To minimize Polaris' time-to-first-paint, future versions of Polaris will render HTML in smaller increments; this will not affect Polaris' ability to optimize network utilization.

5.3 Browser Caching

Up to this point, our experiments have used cold browser caches. In this section, we evaluate Polaris' performance when caches are warm. To do so, we examined the HTTP headers in our recorded web pages, and, for each object that was marked as cacheable, we rewrote the caching headers to ensure that the object would remain cacheable for the duration of our experiment. Then, for each page, we cleared the browser's cache, and loaded the page twice, recording the elapsed time for the second load.

Figure 10 depicts Polaris' benefits with warm caches; the improvements are normalized with respect to Po-

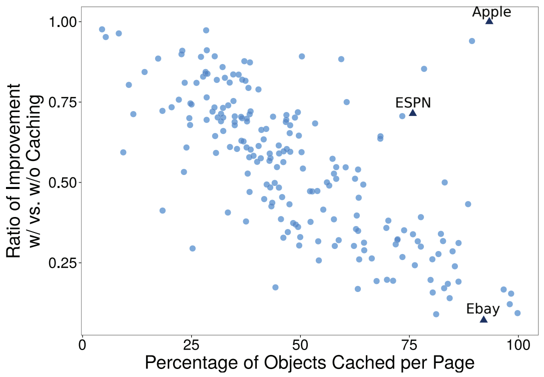


Figure 10: Polaris' benefits with warm caches, normalized with respect to Polaris' gains with cold caches. Each data point represents one of the 200 sites in our corpus. Pages were loaded over a 12 Mbits/s link with an RTT of 100 ms.

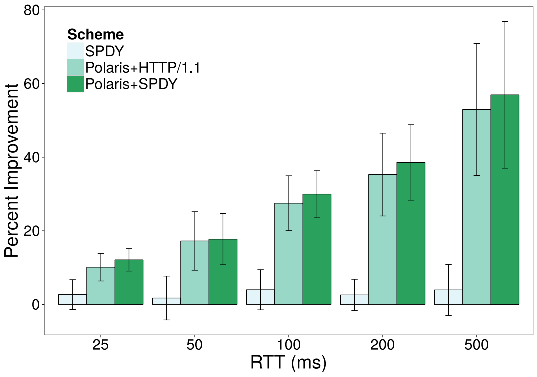


Figure 11: Average reductions in page load time using SPDY, Polaris over HTTP/1.1, and Polaris over SPDY. The performance baseline was load time using HTTP/1.1. The link rate was 12 Mbits/s.

laris' improvements when caches are cold. In general, Polaris' benefits decrease as cache hit rates increase, because there are fewer opportunities for Polaris to optimize network fetches. For example, Ebay caches 92% of all objects, including most of the JavaScript files involved in deep dependency chains; thus, Polaris provides little advantage over the standard scheduling algorithm.

That being said, there are many instances in which caching does not touch objects along a page's critical path. For example, on ESPN's site, 76% of objects are cacheable, but only one object on the deepest dependency chain is cached. Furthermore, a.espncdn.com serves many uncacheable images and JavaScript objects, leading Firefox's standard scheduler to bury critical path JavaScript files behind images that are not on the critical path (§5.2). So, even though ESPN caches 76% of its objects, Polaris still provides 71% of its cold-cache benefits.

Note that the Apple site is an outlier: it caches 93% of its objects, but Polaris provides little benefit in the cold cache case (§5.2), so Polaris provides most of that negligible benefit in the warm cache case as well.

5.4 SPDY

Google proposed SPDY [22], a transport protocol for HTTP messages, to remedy several problems with the HTTP/1.1 protocol. SPDY differs from HTTP/1.1 in four major ways:

- First, SPDY uses a single TCP connection to multiplex all of a browser's HTTP requests and responses involving a particular origin. This allows HTTP requests to be pipelined, and reduces the TCP and TLS handshake overhead that would be incurred if a browser opened multiple TCP connections to an origin.
- SPDY also allows a browser to prioritize the fetches of certain objects (e.g., JavaScript files which block HTML parsing). Priorities give servers hints about how to allocate limited bandwidth to multiple responses.
- SPDY compresses HTTP headers. HTTP is a text-based protocol, so compression can result in non-trivial bandwidth savings.
- Finally, SPDY allows a server to proactively push objects to a browser if the server believes that the browser will request those objects in the near future.

SPDY was a major influence on the HTTP/2 protocol [4] whose deployment is currently starting.

Mahimahi supports SPDY page loads using the `mod_spdy` Apache extension [20]. Thus, we could use Mahimahi to explore how SPDY interacts with Polaris. We loaded each page in our test corpus using four different schemes: HTTP/1.1 (which all of our previous experiments used), Polaris over HTTP/1.1, SPDY, and Polaris over SPDY. In our experiments, SPDY used TCP multiplexing, object prioritization, and HTTP header compression, but not server push, since few of the sites in our test corpus defined SPDY push policies.

Figure 11 compares load times using the four schemes on a 12 Mbits/s link with various RTTs; the performance baseline is the load time using HTTP/1.1. On average, load times using SPDY are 1.74%–3.98% faster than those with HTTP/1.1. Load times using Polaris over SPDY are 2.05%–4.03% faster than those with Polaris over HTTP/1.1. These results corroborate prior work [38] which found that object dependencies limit the ability of SPDY to maximize network utilization. For example, a SPDY-enabled browser may prioritize a JavaScript file in hopes of minimizing the stall time of the HTML parser. However, without Polaris, the SPDY-enabled browser is still limited by conservative lexical dependencies (§2.2), meaning that it cannot aggressively fetch objects "out-of-order" with respect to lexical constraints. In contrast, both Polaris over HTTP/1.1 and Polaris over SPDY have fine-grained dependency information. That information allows Polaris to issue out-of-lexical-order fetches which reduce page load time while respecting the page's intended data flow semantics.

In theory, SPDY-enabled web servers could use Scout's dependency graphs to guide server push policies. However, we believe that *clients*, not servers, are best qualified to make decisions about how a client's network pipe should be used. A server from origin `X` cannot see the objects being pushed by origin `Y`, so different origins may unintentionally overload a client's resource-constrained network connection. Furthermore, Scout's dependency graphs do not capture dynamic critical paths, i.e., the set of object fetches which a client should prioritize *at the current moment* (§4). Thus, a well-intentioned server may *hurt* load time by pushing objects which are not on a dynamic critical path. Polaris avoids this problem using dynamic client-side scheduling.

6 Related Work

Prior dependency trackers [8, 13, 25, 26, 37] deduce dependencies using lexical relationships between HTML tags. As discussed in Sections 2.2 and 3.3, those lexical relationships do not capture fine-grained data flows. As a result, load schedulers which use those dependency graphs are forced to make conservative assumptions to preserve correctness.

WebProphet [18] determines the dependencies between objects by carefully perturbing network fetch delays for individual objects; delaying a parent should delay the loads of dependent children. This technique also relies on course-grained lexical dependencies, since the perturbed browser uses those HTML dependencies to determine which objects to load.

Silo [23] uses aggressive inlining of JavaScript and CSS to fetch entire pages in one or two RTTs. However, Silo does not use the CPU and the network in parallel—all content is fetched, and then all content is evaluated. In contrast, Polaris overlaps computation with network fetches.

Compression proxies like Google FlyWheel [1] and Opera Turbo [29] transparently compress objects before transmitting them to clients. For example, FlyWheel re-encodes images into space-saving formats, and minifies JavaScript and CSS. Polaris is complementary to such techniques.

JavaScript module frameworks like RequireJS [7] and ModuleJS [16] allow developers to manually specify dependencies between JavaScript libraries. Once the dependencies are specified, the frameworks ensure that the relevant libraries are loaded in the appropriate order. Keeping manually-specified dependencies up-to-date can be challenging for a large web site. In contrast, Scout *automatically* tracks fine-grained dependencies between JavaScript files. Scout also tracks dependencies involving HTML, CSS, and images.

7 Conclusion

Prior approaches for loading web pages have been constrained by uncertainty. The objects in a web page can interact in complex and subtle ways; however, those subtle interactions are only partially captured by lexical relationships between HTML tags. Unfortunately, prior load schedulers have used those lexical relationships to extract dependency graphs. The resulting graphs are underspecified and omit important edges. Thus, load schedulers which use those graphs must be overly conservative, to preserve correctness in the midst of hidden dependencies. The ultimate result is that web pages load more slowly than necessary.

In this paper, we use a new tool called Scout to track the fine-grained data flows that arise during a page's load process. Compared to traditional dependency trackers, Scout detects 30% more edges for the median page, and 118% more edges for the 95th percentile page. These additional edges actually give browsers *more* opportunities to reduce load times, because they enable more aggressive fetch schedules than allowed by conservative, lexically-derived dependency graphs. We introduce a new client-side scheduler called Polaris which leverages Scout graphs to assemble a page. By prioritizing the fetches of objects along the dynamic critical path, Polaris minimizes the number of RTTs needed to load a page. Experiments with real pages and varied network conditions show that Polaris reduces load times by 34% for the median page, and 59% for the 95th percentile page.

8 Acknowledgements

We thank Katrina LaCurts, Amy Ousterhout, the NSDI reviewers, and our shepherd, Paul Barford, for their helpful comments and suggestions. We also thank the members of the MIT Center for Wireless Networks and Mobile Computing (Wireless@MIT) for their support. This work was supported in part by NSF grant CNS-1407470.

References

[1] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *Proceedings of NSDI*, 2015.

[2] Alexa. Top Sites in United States. http://www.alexa.com/topsites/countries/US, 2015.

[3] Amazon. Silk Web Browser. https://amazonsilk.wordpress.com/, December 16, 2014.

[4] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2. http://httpwg.org/specs/rfc7540.html, May 2015.

[5] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-perceived Quality into Web Server Design. In *Proceedings of World Wide Web Conference on Computer Networks: The International Journal of Computer and Telecommunications Networking*, 2000.

[6] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. In *Proceedings of CHI*, 2000.

[7] J. Burke. RequireJS. http://requirejs.org/, 2015.

[8] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of NSDI*, 2015.

[9] Chromium. web-page-replay. https://github.com/chromium/web-page-replay, 2015.

[10] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web Site Delays: How Tolerant are Users? *Journal of the Association for Information Systems*, 5(1), 2004.

[11] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-Scale Control Plane for Video Quality Optimization. In *Proceedings of NSDI*, 2015.

[12] Google. Using site speed in web search ranking. http://googlewebmastercentral.blogspot.com/2010/04/using-site-speed-in-web-search-ranking.html, April 9, 2010.

[13] Google. Chrome DevTools Overview. https://developer.chrome.com/devtools, August 2013.

[14] A. Hidayat. Esprima. http://esprima.org, 2015.

[15] J. Jiang, V. Sekar, and H. Zhang. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *Proceedings of CoNext*, 2012.

[16] L. Jung. modulejs lightweight JavaScript module system. https://larsjung.de/modulejs/, 2016.

[17] Q. Li, W. Zhou, M. Caesar, and P. B. Godfrey. ASAP: A Low-latency Transport Layer. In *Proceedings of SIGCOMM*, 2011.

[18] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: Automating Performance Prediction for Web Services. In *Proceedings of NSDI*, 2010.

[19] Google Developers. Remove Render-Blocking JavaScript. https://developers.google.com/speed/docs/insights/BlockingJS, April 8, 2015.

[20] Google Developers. SPDY. https://developers.google.com/speed/spdy/mod_spdy/, May 27, 2015.

[21] The Chromium Projects. QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic, 2015.

[22] The Chromium Projects. SPDY. https://www.chromium.org/spdy, 2015.

[23] J. Mickens. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. In *Proceedings of WebApps*, 2010.

[24] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of NSDI*, 2010.

[25] Microsoft. Meet the Microsoft Edge Developer Tools. https://dev.windows.com/en-us/microsoft-edge/platform/documentation/f12-devtools-guide/, 2015.

[26] Mozilla. Firefox Developer Tools. https://developer.mozilla.org/en-US/docs/Tools, 2015.

[27] Mozilla. Proxy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy, February 16, 2016.

[28] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of USENIX ATC*, 2015.

[29] Opera. Data savings and turbo mode. http://www.opera.com/turbo, 2015.

[30] Opera. Opera Mini. http://www.opera.com/mobile/mini, 2015.

[31] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proceedings of CoNext*, 2011.

[32] L. Richardson. Beautiful Soup. http://www.crummy.com/software/BeautifulSoup/, February 17, 2016.

[33] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. PARCEL: Proxy Assisted BRowsing in Cellular Networks for Energy and Latency Reduction. In *Proceedings of CoNext*, 2014.

[34] S. Sundaresan, N. Feamster, R. Teixeira, and N. Magharei. Measuring and Mitigating Web Performance Bottlenecks in Broadband Access Networks. In *Proceedings of IMC*, 2013.

[35] Y. Suzuki. Escodegen. https://github.com/estools/escodegen, 2015.

[36] Y. Suzuki. Estraverse. https://github.com/estools/estraverse, 2016.

[37] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of NSDI*, 2013.

[38] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *Proceedings of NSDI*, 2014.

CFA: A Practical Prediction System for Video QoE Optimization

Junchen Jiang[†], *Vyas Sekar*[†], *Henry Milner*[⋆], *Davis Shepherd*[+], *Ion Stoica*[⋆+◦], *Hui Zhang*[†+]
[†]*CMU*, [⋆]*UC Berkeley*, [+]*Conviva*, [◦]*Databricks*

Abstract

Many prior efforts have suggested that Internet video Quality of Experience (QoE) could be dramatically improved by using data-driven prediction of video quality for different choices (e.g., CDN or bitrate) to make optimal decisions. However, building such a prediction system is challenging on two fronts. First, the relationships between video quality and observed session features can be quite complex. Second, video quality changes dynamically. Thus, we need a prediction model that is (a) expressive enough to capture these complex relationships and (b) capable of updating quality predictions in near real-time. Unfortunately, several seemingly natural solutions (e.g., simple machine learning approaches and simple network models) fail on one or more fronts. Thus, the potential benefits promised by these prior efforts remain unrealized. We address these challenges and present the design and implementation of Critical Feature Analytics (CFA). The design of CFA is driven by domain-specific insights that video quality is typically determined by a small subset of critical features whose criticality persists over several tens of minutes. This enables a scalable and accurate workflow where we automatically learn critical features for different sessions on coarse-grained timescales, while updating quality predictions in near real-time. Using a combination of a real-world pilot deployment and trace-driven analysis, we demonstrate that CFA leads to significant improvements in video quality; e.g., 32% less buffering time and 12% higher bitrate than a random decision maker.

1 Introduction

Delivering high quality of experience (QoE) is crucial to the success of today's subscription and advertisement-based business models for Internet video. As prior work (e.g., [33, 11]) has shown, achieving good QoE is challenging because of significant spatial and temporal variation in CDNs' performance, client-side network conditions, and user request patterns.

At the same time, these observations also suggest there is a substantial room for improving QoE by dynamically selecting the optimal CDN and bitrate based on a real-time global view of network conditions. Building on this insight, prior work makes the case for a quality optimization system (Figure 1) that uses a *prediction oracle* to suggest the best parameter settings (e.g., bitrate, CDN) to optimize quality (e.g., [33, 11, 35, 32, 20]). Seen in a broader context, this predictive approach can be applied beyond Internet video (e.g., [10, 40, 15, 16, 43]).

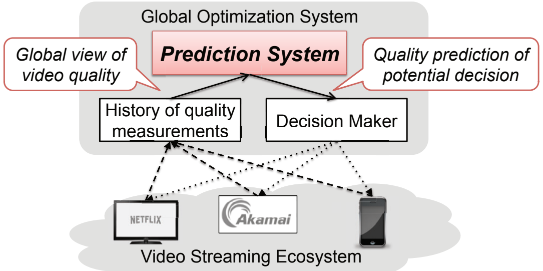

Global Optimization System
Global view of video quality
Prediction System
Quality prediction of potential decision
History of quality measurements
Decision Maker
NETFLIX
Akamai
Video Streaming Ecosystem

Figure 1: ***Overview of a global optimization system and the crucial role of a prediction system.***

However, these prior efforts fall short of providing a concrete instantiation of such a prediction system. Specifically, we observe that designing such a prediction system is challenging on two key fronts (§2):

- **Capturing complex factors that affect quality:** For instance, an outage may affect only clients of a specific ISP in a specific city when they use a specific CDN. To accurately predict the quality of their sessions, one must consider the combination of all three factors. In addition, the factors that affect video quality vary across different sessions; e.g., wireless hosts may be bottlenecked at the last connection, while other clients may experience loading failures due to unavailability of specific content on some CDNs.
- **Need for fresh updates:** Video quality changes rapidly, on a timescale of several minutes. Ideally, we must make predictions based on recent quality measurements. This is particularly challenging given the volume of measurements (e.g., YouTube had 231 million video sessions and up to 500 thousand concurrent viewers during the Olympics [7]), compounded with the need for expressive and potentially complex prediction models.

Unfortunately, many existing solutions fail on one or both counts. For instance, solutions that use less complex models (e.g., linear regression, Naive Bayes, or simple models based on last-mile connection) are not expressive enough to capture high dimensional and diverse relationships between video quality and session features. More complex algorithms (e.g., SVM [42]) can take several hours to train a prediction model and will be inaccurate because predictions will rely on stale data.

In this work, we address these challenges and present the design and implementation of a quality prediction system called *Critical Feature Analytics* (CFA). CFA is built on three key domain-specific insights:

1. Video sessions with *same feature values* have *similar quality*. This naturally leads to an expressive model, wherein the video quality of a given session can be accurately predicted based on the quality of sessions that match values on all features (same ASN, CDN, player, geographical region, video content, etc). However, if applied naively, this model can suffer from the curse of dimensionality — as the number of combinations of feature values grows, it becomes hard to find enough matching sessions to make reliable predictions.
2. Each video session has a *subset* of *critical features* that ultimately determines its video quality. Given this insight, we can make more reliable predictions based on similar sessions that only need to match on critical features. For example, in a real event that we observed, congestion of a Level3 CDN led to relatively high loading failure rate for Comcast users in Baltimore. We can accurately predict the quality of the affected sessions using sessions associated with the specific CDN, region and ISP, ignoring other non-critical features (e.g., player, video content). Thus, this tackles the curse of dimensionality, while still retaining sufficient expressiveness for accurate prediction (§3).
3. Critical features tend to be *persistent*. Two remaining concerns are: (a) Can we identify critical features and (b) How expensive is it to do so? The insight on persistence implies that critical features are learnable from recent history and can be cached and reused for fast updates (§4). This insight is derived from recent measurement studies [25, 20] (e.g., the factors that lead to poor video quality persist for hours, and sometimes, even days).

Taken together, these insights enable us to engineer a scalable and accurate video quality prediction system. Specifically, on a coarse timescale of tens of minutes, CFA learns the critical features, and on a fine timescale of minutes, CFA updates quality prediction using recent quality measurements. CFA makes predictions and decisions as new clients arrive.

We implemented a prototype of CFA and integrated it in a video optimization platform that manages many premium video providers. We ran a pilot study on one content provider that has 150,000 sessions each day. Our real-world experiments show that the bitrates and CDNs selected by CFA lead to 32% less buffering time and 12% higher bitrate than a baseline random decision maker. Using real trace-driven evaluation, we also show that CFA outperforms many other simple ML prediction algorithms by up to 30% in prediction accuracy and 5-17% in various video quality metrics.

Contributions and Roadmap:

- Identifying key challenges in building an accurate prediction system for video quality (§2).
- Design and implementation of CFA, built on domain-specific insights to address the challenges (§3-5).
- Real-world and trace-driven evaluation that demonstrates substantial quality improvement by CFA (§6).
- Using critical features learned by CFA to make interesting observations about video quality (§7).

2 Background and Challenges

This section begins with some background on video quality prediction (§2.1). Then, we articulate two key challenges faced by any video quality prediction system: (1) The factors affecting video quality are *complex*, so we need expressive models (§2.2); (2) Quality changes *rapidly*, so models must be updated in near real-time by recent quality measurements (§2.3). We also argue why existing solutions do not address these challenges.

2.1 Background

Most video service providers today allow a video client (player) to switch CDN and bitrate among a set of available choices [33, 20, 32]. These switches have little overhead and can be performed at the beginning of and during a video playback [8]. Our goal then is to choose the best CDN and bitrate for a client by *accurately predicting the video quality of each hypothetical choice of CDN and bitrate*. In theory, if we can accurately predict the quality of each potential decision, then we can identify the optimal decision.

To this end, we envision a prediction system that uses a *global view* of quality measurements to make predictions for a specific video session. It learns a *prediction function* for each quality metric $Pred : 2^{\mathbb{S}} \times \mathbb{S} \mapsto \mathbb{R}$, which takes as input a given set of historical sessions $S \in 2^{\mathbb{S}}$ whose quality is already measured, and a new session $s \in \mathbb{S}$, and outputs a quality prediction $p \in \mathbb{R}$ for s.

Each quality measurement summarizes the quality of a video session for some duration of time (in our case, one minute). It is associated with values of four *quality metrics* [18] and a set of *features*[2] (summarized in Table 1).

[1] For one session, VSF is zero if it starts successfully, one otherwise.

[2] By feature, we refer to the type of attribute (e.g., *CDN*), rather than value of these attributes (e.g., $CDN = Akamai$)

Metrics	Description
BufRatio	Fraction of time a session spends in buffering (smooth playback is interrupted by buffering).
AvgBitrate	Time-weighted average of bitrates in a session.
JoinTime	Delay for the video to start playing from the time the user clicks "play".
Video start failure (VSF)	Fraction of sessions that fail to start playing (e.g., unavailable content or overloaded server)[1].
Features	**Description**
ASN	Autonomous System to which client IP belongs.
City	City where the client is located.
ConnectionType	Type of access network; e.g., mobile/fixed wireless, DSL, fiber-to-home [3].
Player	e.g., Flash, iOS, Silverlight, HTML5.
Site	Content provider of requested video contents.
LiveOrVoD	Binary indicator of live vs. VoD content.
ContentName	Name of the requested video object.
CDN	CDN a session started with.
Bitrate	Bitrate value the session started at.

Table 1: ***Quality metrics and session features associated with each session. CDN and Bitrate refer to initial CDN/bitrate values as we focus on initial selections.***

In general, the set of features depends on the degree of instrumentation and what information is visible to a specific provider. For instance, a CDN may know the location of servers, whereas a third-party optimizer [1] may only have information at the CDN granularity. Our focus is not to determine the best set of features that should be recorded for each session, but rather engineer a prediction system that can take an arbitrary set of features as inputs and extract the relationships between these features and video quality. In practice, the above set of features can already provide accurate predictions that help improve quality.

Our dataset consists of 6.6 million quality measurements collected from 2 million clients using 3 large public CDNs distributed across 168 countries and 152 ISPs.

2.2 Challenge 1: Expressive models

We show real examples of the complex factors that impact video quality, and the limitations in capturing these relationships.

High-dimensional relationship between video quality and session features. Video quality could be impacted by *combinations* of multiple components in the network. Such high-dimensional effects make it harder to learn the relationships between video quality and features, in contrast to simpler settings where features affect quality independently (e.g., assumed by Naive Bayes).

In a real-world incident, video sessions of Comcast users in Baltimore who watched videos from Level3 CDN experienced high failure rate (VSF) due to congested edge servers, shown by the blue line in Figure 2. The figure also shows the VSF of sessions sharing the same values on one or two features with the affected sessions; e.g., all Comcast sessions across different cities and CDNs. In the figure, the high VSF of the affected sessions cannot be clearly identified if we look at the sessions that match on only one or two features. Only when three features of *CDN* ("Level3"), *ASN* ("Comcast") and *City* ("Baltimore") are specified (i.e., blue line), can we detect the high VSF and predict the quality of affected sessions accurately.

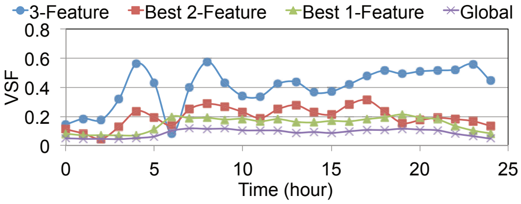


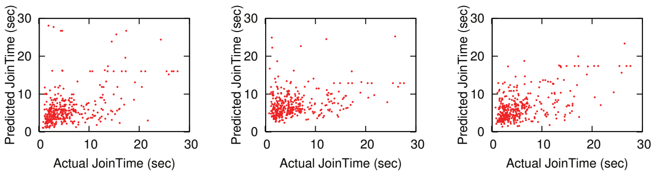
Figure 2: ***The high VSF is only evident when three factors (CDN, ISP and geo-location) are combined.***

In practice, we find that such high-dimensional effects are the common case, rather than an anomalous corner case. For instance, more than 65% of distinct CDN-ISP-City values have VSF that is at least 50% higher or lower than the VSF of sessions matching only one or two features (not shown). In other words, their quality is affected by a combined effect of at least three features.

Limitation of existing solutions: It might be tempting to develop simple predictors; e.g., based on the last-hop connection by using average quality of history sessions with the same *ConnectionType* value. However, they do not take into account the combined impact of features on video quality. Conventional machine learning techniques like Naive Bayes also suffer from the same limitation. In Figures 3(a) and 3(b), we plot the actual JoinTime and the prediction made by the last-hop predictor and Naive Bayes (from `Weka` [6]) for 300 randomly sampled sessions. The figures also show the mean relative error ($\frac{|predicted-actual|}{actual}$). For each session, the prediction algorithms train models using historical sessions within a 10-minute interval prior to the session under prediction. It shows that the prediction error of both solutions is significant and two-sided (i.e., not fixable by normalization).

Highly diverse structures of factors. The factors that affect video quality vary across different sessions. This means the prediction algorithm should be expressive enough to predict quality for different sessions using different prediction models. For instance, the fact that many fiber-to-the-home (e.g., FiOS) users have high bitrates and people on cellular connections have lower bitrates is largely due to the speed of their last-mile connection. In contrast, some video clients may experience video loading failures due to unavailability of specific content on some CDNs. A recent measurement study [25] has shown that many heterogeneous factors are correlated with video quality issues. In §7, we show that 15% of video sessions are impacted by more than 30 different combinations of features and give real examples of different factors that affect quality.

Limitation of existing solutions: To see why existing solutions are not sufficient, let us consider the *k*-nearest



(a) Last hop (0.76) (b) Naive Bayes (0.61) (c) *k*-NN (0.63)

Figure 3: ***Prediction error of some existing solutions is substantial (mean of relative error in parentheses).***

neighbor (*k*-NN) algorithm. It does not handle diverse relationships between quality and features, because the similarity between sessions is based on the same function of features independent of the specific session under prediction. In Figure 3(c), we plot the actual values of JoinTime and the prediction made by *k*-NN with the same setup as Figure 3(a)(b). Similar to Naive Bayes and the last-hop predictor, *k*-NN has substantial prediction error.

2.3 Challenge 2: Fresh updates

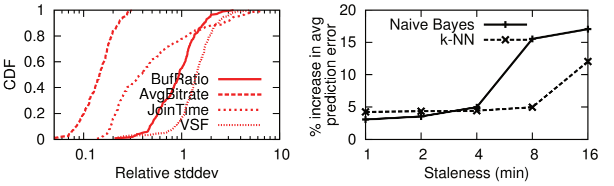
Video quality has significant temporal variability. In Figure 4(a), for each quality metric and combination of specific CDN, city and ASN, we compute the mean quality of sessions in each 10-minute interval, and then plot the CDF of the relative standard deviation ($\frac{stddev}{mean}$) of the quality across different intervals. In all four quality metrics of interest, we see significant temporal variability; e.g., for 60% of CDN-city-ASN combinations, the relative standard deviation of JoinTime across different 10-minute intervals is more than 30%. Such quality variability has also been confirmed in other studies (e.g., [33]).

The implication of such temporal variability is that the prediction system must update models in near real-time. In Figure 4(b), we use the same setup as Figure 3, except that the time window used to train prediction models is several minutes prior to the session under prediction. The figure shows the impact of such staleness on the prediction error for JoinTime. For both algorithms, prediction error increases dramatically if the staleness exceeds 10 minutes. As we will see later, this negative impact of staleness on accuracy is not specific to these prediction algorithms (§6.3).

Limitation of existing solutions: The requirement to use the most recent measurements makes it infeasible to use computationally expensive models. For instance, it takes at least one hour to train an SVM-based prediction model from 15K quality measurements in a 10-minute interval for one video site, so the quality predictions will be based on information from more than one hour ago.

3 Intuition behind CFA

This section presents the domain-specific insights we use to help address the expressiveness challenge (§2.2). The first insight is that sessions matching on all features have similar video quality. However, this approach suffers from the curse of dimensionality. Fortunately, we can leverage a second insight that each video session has a subset of *critical features* that ultimately determine its video quality. We conclude this section by highlighting two outstanding issues in translating these insights into a practical prediction system.



(a) Temporal variability (b) Impact of staleness on accuracy

Figure 4: ***Due to significant temporal variability of video quality (left), prediction error increases dramatically with stale data (right).***

```
Input: Session under prediction $s$, Previous sessions $S$
Output: Predicted quality $p$
/* $S'$:identical sessions matching on all
   features with $s$ in recent history($\Delta$)     */
$S' \leftarrow SimilarSessionSet(s, S, AllFeatures, \Delta)$;
/* Summarize the quality (e.g.,median) of
   the identical sessions in $S'$.                   */
$p \leftarrow Est(S')$;
return $p$;
```

Algorithm 1: ***Baseline prediction that finds sessions matching on all features and uses their observed quality as the basis for prediction.***

3.1 Baseline prediction algorithm

Our first insight is that sessions that have identical feature values will naturally have similar (if not identical) quality. For instance, we expect that all Verizon FiOS users viewing a specific HBO video using Level3 CDN in Pittsburgh at Fri 9 am should have similar quality (modulo very user-specific effects such as local Wi-Fi interference inside the home). We can summarize the intuition as follows:

Insight 1: ***At a given time, video sessions having same value on every feature have similar video quality.***

Inspired by Insight 1, we can consider a baseline algorithm (Algorithm 1). We predict a session's quality based on "identical sessions", i.e., those from recent history that match values on *all* features with the session under prediction. Ideally, given infinite data, this algorithm is accurate, because it can capture all possible combinations of factors affecting video quality.

However, this algorithm is unreliable as it suffers from the classical curse of dimensionality [39]. Specifically, given the number of combinations of feature values (ASN, device, content providers, CDN, just to name a few), it is hard to find enough identical sessions needed to make a robust prediction. In our dataset, more than 78% of sessions have no identical session (i.e., matching on all features) within the last 5 minutes.

3.2 Critical features

In practice, we expect that some features are more likely to "explain" the observed quality of a specific video session than others. For instance, if a specific peering point between Comcast and Netflix in New York is congested, then we expect most of these users will suffer poor quality, regardless of the speed of their local connection.

Insight 2: ***Each video session has a subset of* critical features *that ultimately determines its video quality.***

We already saw some real examples in §2.2: in the example of high dimensionality, the critical features of the sessions affected by the congested Level3 edge servers are $\{ASN, CDN, City\}$; in the examples of diversity, the critical features are $\{ConnectionType\}$ and $\{CDN, ContentName\}$. Table 2 gives more real examples of critical features that we have observed in operational settings and confirmed with domain experts.

Quality issue	**Set of critical features**
Issue on one player of Vevo	$\{Player, Site\}$
ESPN flipping between CDNs	$\{CDN, Site, ContentName\}$
Bad Level3 servers for Comcast users in Maryland	$\{CDN, City, ASN\}$

Table 2: *Real-world examples of critical features confirmed by analysts at a large video optimization vendor.*

A natural implication of this insight is that it can help us tackle the curse of dimensionality. Unlike Algorithm 1, which fails to find a sufficient number of sessions, we can estimate quality more reliably by aggregating observations across a larger amount of "similar sessions" that only need to match on these *critical features*. Thus, critical features can provide expressiveness while avoiding curse of dimensionality.

Algorithm 2 presents a logical view of this idea:

1. **Critical feature learning (line 1):** First, find the critical features of each session s, denoted as $CriticalFeatures(s)$.
2. **Quality estimation (line 2, 3):** Then, find similar sessions that match values with s on critical features $CriticalFeatures(s)$ within a recent history of length Δ (by default, 5 minutes). Finally, return some suitable estimate of the quality of these similar sessions; e.g., the median[3] (for BufRatio, AvgBitrate, JoinTime) or the mean (for VSF).

A practical benefit of Algorithm 2 is that it is interpretable [52], unlike some machine learning algorithms (e.g., PCA or SVM). This allows domain experts to combine their knowledge with CFA and diagnose prediction errors or resolve incidents, as we explore in §7.2.

```
  Input: Session under prediction $s$, Previous sessions $S$
  Output: Predicted quality $p$
  /* $CF_s$:Set of critical features of $s$                */
1 $CF_s \leftarrow CriticalFeatures(s)$;
  /* $S'$:Similar sessions matching values on
     critical features $CF_s$ with $s$.                    */
2 $S' \leftarrow SimilarSessionSet(s, S, CF_s, \Delta)$;
  /* Summarize the quality of the similar
     sessions in $S'$.                                      */
3 $p \leftarrow Est(S')$;
4 return $p$;
```

Algorithm 2: *CFA prediction algorithm, where prediction is based on similar sessions matching on critical features.*

At this time, it is useful to clarify what critical features are and what they are not. In essence, critical features provide the explanatory power of how a prediction is made. However, critical features are not a minimal set of factors that determine the quality (i.e., root cause). That is, they can include both features that reflect the root cause as well as additional features. For example, if all HBO sessions use Level3, their critical features may include both CDN and $Site$, even if CDN is redundant, since including it does not alter predictions. The primary objective of CFA is accurate prediction; root cause diagnosis may be an added benefit.

3.3 Practical challenges

There are two issues in using Algorithm 2.

Can we learn critical features? A key missing piece is how we get the critical features of each session (line 1). This is challenging because critical features vary both across sessions and over time [33, 25], and it is infeasible to manually configure critical features.

How to reduce update delay? Recall from §2.3 that the prediction system should use the most recent quality measurements. This requires a scalable implementation of Algorithm 2, where critical features and quality estimates are updated in a timely manner. However, naively running Algorithm 2 for millions of sessions under prediction is too expensive (§6.3). With a cluster of 32 cores, it takes 30 minutes to learn critical features for 15K sessions within a 10-minutes interval. This means the prediction will be based on stale information from tens of minutes ago.

4 CFA Detailed Design

In this section, we present the detailed design of CFA and discuss how we address the two practical challenges mentioned in the previous section: learning critical features and reducing update delay.

[3] We use median because it is more robust to outliers.

Notations	Domains	Definition
$s,S,\mathbb{S}$		A session, a set of sessions, set of all sessions
$q(s)$	$\mathbb{S}\mapsto\mathbb{R}$	Quality of s
$QualityDist(S)$	$2^{\mathbb{S}}\mapsto 2^{\mathbb{R}}$	$\{q(s) \mid s\in S\}$
$f,F,\mathbb{F}$		A feature, a set of features, set of all features
$CriticalFeatures(s)$	$\mathbb{S}\mapsto 2^{\mathbb{F}}$	Critical features of s
$\mathbb{V}$		Set of all feature values
$FV(f,s)$	$\mathbb{F}\times\mathbb{S}\mapsto\mathbb{V}$	Value on feature f of s
$FSV(F,s)$	$2^{\mathbb{F}}\times\mathbb{S}\mapsto 2^{\mathbb{V}}$	Set of values on features in F of s
$SimilarSessionSet$ (s,S,F,Δ)	$\mathbb{F}\times 2^{\mathbb{F}}\times\mathbb{S}\times$ $\mathbb{R}^{+}\mapsto 2^{\mathbb{F}}$	$\{s' \mid s'\in S, t(s)-\Delta < t(s') < t(s), FSV(F,s') = FSV(F,s)\}$

Table 3: ***Notations used in learning of critical features.***

The key to addressing these challenges is our third and final domain-specific insight:

Insight 3: ***Critical features tend to* persist *on long timescales of tens of minutes.***

This insight is derived from prior measurement studies [25, 20]. For instance, our previous study on shedding light on video quality issues in the wild showed that the factors that lead to poor video quality persist for hours, and sometimes even days [25]. Another recent study from the C3 system suggests that the best CDN tends to be relatively stable on the timescales of few tens of minutes [20]. We independently confirm this observation in §6.3 that using slightly stale critical features (e.g., 30-60 minutes ago) achieves similar prediction accuracy as using the most up-to-date critical features. Though this insight holds for most cases, it is still possible (e.g., on mobile devices) that critical features persist on a relatively shorter timescale (e.g., due to the nature of mobility).

Note that the persistence of critical features does not mean that quality values are equally persistent. In fact, persistence of critical features is on a timescale an order of magnitude longer than the persistence of quality. That is, even if quality fluctuates rapidly, the critical features that determine the quality do not change as often.

As we will see below, this persistence enables (a) automatic learning of critical features from history, and (b) a scalable workflow that provides up-to-date estimates.

4.1 Learning critical features

Recall that the first challenge is obtaining the critical features for each session. The persistence of critical features has a natural corollary that we can use to automatically learn them:

Corollary 3.1: ***Persistence implies that critical features of a session are learnable from history.***

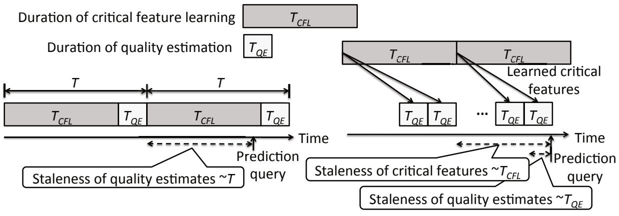
Specifically, we can simply look back over the history and identify the subset of features F such that the quality distribution of sessions matching on F is most similar to that of sessions matching on *all* features. For instance, suppose we have three features $\langle ContentName, ASN, CDN\rangle$ and it turns out that sessions with $ASN = Comcast, CDN = Level3$ consistently have high buffering over the last few hours due to some internal congestion at the corresponding exchange point. Then, if we look back over the last few hours, the data from history will naturally reveal that the distribution of the quality of sessions with the feature values $\langle ContentName = Foo, ASN = Comcast, CDN = Level3\rangle$ will be similar to $\langle ContentName = *, ASN = Comcast, CDN = Level3\rangle$, but very different from, say, the quality of sessions in $\langle ContentName = *, ASN = *, CDN = Level3\rangle$, or $\langle ContentName = *, ASN = Comcast, CDN = *\rangle$. Thus, we can use a data-driven approach to learn that ASN, CDN are the critical features for sessions matching $\langle ContentName = Foo, ASN = Comcast, CDN = Level3\rangle$.

```
Input: Session under prediction $s$, Previous sessions $S$
Output: Critical features for $s$
/* Initialization                                        */
1  $MaxSimilarity \leftarrow -\infty, CriticalFeatures \leftarrow NULL$;
/* $D_{finest}$:Quality distribution of
   sessions matching on $\mathbb{F}$ in $\Delta_{learn}$.   */
2  $D_{finest} \leftarrow QualityDist(SimilarSessionSet(s,S,\mathbb{F},\Delta_{learn}))$;
3  for $F \subseteq 2^{\mathbb{F}}$ do
       /* Exclude $F$ without enough similar
          sessions for prediction.                        */
4      if $|SimilarSessionSet(s,S,F,\Delta)| < n$ then
5          continue;
       /* $D_F$:Quality distribution of
          sessions matching on $F$ in $\Delta_{learn}$.
          */
6      $D_F \leftarrow QualityDist(SimilarSessionSet(s,S,F,\Delta_{learn}))$;
       /* Get similarity of $D_{finest}$ & $D_F$.           */
7      $Similarity \leftarrow Similarity(D_F, D_{finest})$;
8      if $Similarity > MaxSimilarity$ then
9          $MaxSimilarity \leftarrow Similarity$;
10         $CriticalFeatures \leftarrow F$;
11 return $CriticalFeature$;
```

Algorithm 3: ***Learning of critical features.***

Algorithm 3 formalizes this intuition for learning critical features. Table 3 summarizes the notation used in Algorithm 3. For each subset of features F (line 3), we compute the similarity between the quality distribution (D_F) of sessions matching on F and the quality distribution (D_{finest}) of sessions matching on all features (line 7). Then, we find the F that yields the maximum similarity (line 8-10), under one additional constraint that $SimilarSessionSet(s,S,F,\Delta)$ should include enough (by default, at least 10) sessions to get reliable quality estimation (line 4-5). This check ensures that the algorithm will not simply return the set of all features.

As an approximation of the duration in which criti-


Duration of critical feature learning
T_{CFL}
Duration of quality estimation
T_{QE}
T
T
T_{CFL}
T_{QE}
T_{CFL}
T_{QE}
Time
Prediction query
Staleness of quality estimates ~T
T_{CFL}
T_{CFL}
Learned critical features
T_{QE}
T_{QE}
...
T_{QE}
T_{QE}
Time
Prediction query
Staleness of critical features ~T_{CFL}
Staleness of quality estimates ~T_{QE}

(a) Naive workflow (b) CFA workflow

Figure 5: ***To reduce update delay, we run critical feature learning and quality estimation at different timescales by leveraging persistence of critical features.***

cal features persist, we use $\Delta_{learn} = 60min$. Note that Δ_{learn} is an order of magnitude larger than the time window Δ used in quality estimation, because critical features persist on a much longer timescale than quality values. We use (the negative of) Jensen-Shannon divergence between D_1 and D_2 to quantify their similarity $Similarity(D_1, D_2)$.

Although Algorithm 3 can handle most cases, there are corner cases where $SimilarSessionSet(s, S, \mathbb{F}, \Delta_{learn})$ does not have enough sessions (i.e., more than *n*) to compute $Similarity(D_F, D_{finest})$ reliably. In these cases, we replace D_{finest} by the set of *n* sessions that share most features with *s* over the time window of Δ_{learn}. Formally, we use $\{s'|s' \text{ matches } k_s \text{ features with } s\}$, where $k_s = \text{argmin}_k\left(|\{s'|s' \text{ matches } k \text{ features with } s| \geq n\}|\right)$.

4.2 Using fresh updates

Next, we focus on reducing the update delay between when a quality measurement is received and used for prediction.

Naively running critical feature learning and quality estimation of Algorithm 2 can be time-consuming, causing the predictions to rely on stale data. In Figure 5(a), T_{CFL} and T_{QE} are the duration of critical feature learning and the duration of quality estimation, respectively. The staleness of quality estimation (depicted in Figure 5) to respond to a prediction query can be as large as the total time of two steps (i.e., $T_{CFL} + T_{QE}$), which typically is tens of minutes (§6.3). Also, simply using more parallel resources is not sufficient. The time to learn critical features using Algorithm 3 grows linearly with the number of sessions under prediction, the number of history sessions, and the number of possible feature combinations. Thus, the complexity of learning critical features T_{CFL} is exponential in the number of features. Given the current set of features, T_{CFL} is on the scale of tens of minutes.

To reduce update delay, we again leverage the persistence of critical features:

Corollary 3.2: ***Persistence implies that critical features can be cached and reused over tens of minutes.***

Building on Corollary 3.2, we decouple the critical feature learning and quality estimation steps, and run them at separate timescales. On the timescale of tens of minutes, we update the results of critical feature learning. Then, on a faster timescale of tens of seconds, we update quality estimation using fresh data and the most recently learned critical features.

This decoupling minimizes the impact of staleness on prediction accuracy. Learning critical features on the timescale of tens of minutes is sufficiently fast as they persist on the same timescale. Meanwhile, quality estimation can be updated every tens of seconds and makes predictions based on quality updates with sufficiently low staleness. Thus, the staleness of quality estimation T_{QE} of the decoupled workflow (Figure 5(b)) is a magnitude lower than $T_{QE} + T_{CFL}$ of the naive workflow (Figure 5(a)). In §6.3, we show that this workflow can retain the freshness of critical features and quality estimates.

In addition, CFA has a natural property that two sessions sharing all feature values and occurring close in time will map to the same critical features. Thus, instead of running the steps per-session, we can reduce the computation to the granularity of *finest partitions*, i.e., distinct values of all features.

4.3 Putting it together

Building on these insights, we create the following practical *three-stage* workflow of CFA.

- **Stage I: Critical feature learning** (line 1 of Algorithm 2) runs offline, say, every tens of minutes to an hour. The output of this stage is a key-value table called *critical feature function* that maps all observed finest partitions to their critical features.
- **Stage II: Quality estimation** (line 2,3 of Algorithm 2) runs every tens of seconds for all observed finest partitions based on the most recent critical features learned in the first stage. This outputs another key-value table called *quality function* that maps a finest partition to the quality estimation, by aggregating the most recent sessions with the corresponding critical features.
- **Stage III: Real-time query/response.** Finally, we provide real-time query/response on the arrival of each client, operating at the millisecond timescale, by simply looking up the most recent precomputed value function from the previous stage. These operations are simple and can be done very fast.

Finally, instead of forcing all finest partition-level computations to run in every batch, we can do triggered recomputations of critical feature learning only when the observed prediction errors are high.

5 Implementation and Deployment

This section presents our implementation of CFA and highlights engineering solutions to address practical

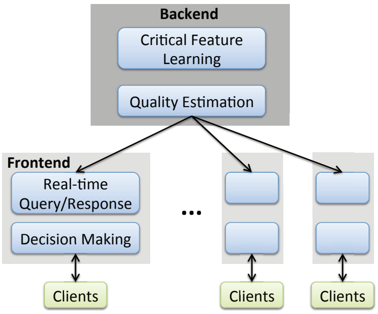


Figure 6: ***Implementation overview of CFA. The three stages of CFA workflow are implemented in a backend cluster and distribute frontend clusters.***

challenges in operational settings (e.g., avoiding bulk data loading and speeding up development iterations).

5.1 Implementation of CFA workflow

CFA's three stages are implemented in two different locations: a centralized backend cluster and geographically distributed frontend clusters as depicted in Figure 6.

Centralized backend: The critical feature learning and quality estimation stages are implemented in a backend cluster as periodic jobs. By default, critical feature learning runs every 30 minutes, and quality estimation runs every minute. A centralized backend is a natural choice because we need a global view of all quality measurements. The quality function, once updated by the estimation step, is disseminated to distributed frontend clusters using Kafka [27].

Note that we can further reduce learning time using simple parallelization strategies. Specifically, the critical features of different finest partitions can be learned independently. Similarly in Algorithm 3, the similarity of quality distributions can be computed in parallel. To exploit this data-level parallelism, we implement them as Spark jobs [4].

Distributed frontend: Real-time query/response and decision makers of CDN/bitrate are co-located in distributed frontend clusters that are closer to clients than the backend. Each frontend cluster receives the quality function from the backend and caches it locally for fast prediction. This reduces the latency of making decisions for clients.

5.2 Challenges in an operational setting

Mitigating impact of bulk data loading: The backend cluster is shared and runs other delay-sensitive jobs; e.g., analytics queries from production teams. Since the critical feature learning runs periodically and loads a large amount of data (≈30 GB), it creates spikes in the delays of other jobs (Figure 7). To address this concern, we engineered a simple heuristic to evenly spread the data retrieval where we load a small piece of data every few minutes. As Figure 7 shows, this reduces the spikes

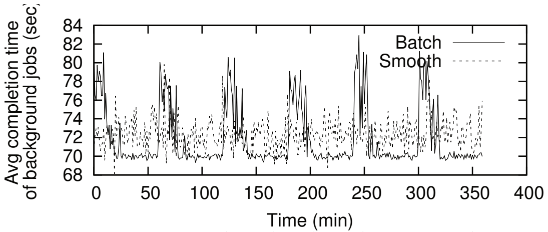
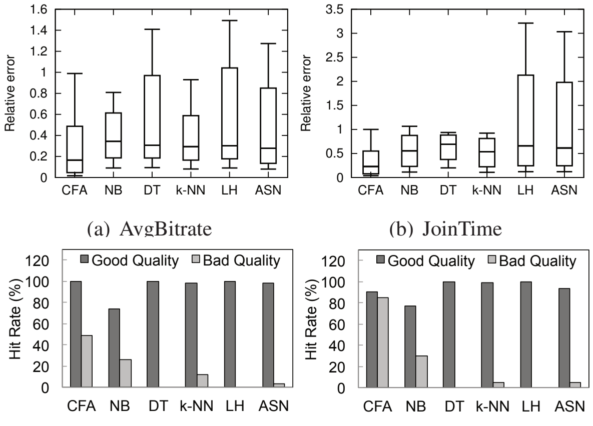


Figure 7: ***Streaming data loading has smoother impact on completion delay than batch data loading.***



(a) AvgBitrate

(b) JoinTime

(c) BufRatio

(d) VSF

Figure 8: ***Distributions of relative prediction error*** *($\{5, 10, 50, 90, 95\}$%**iles) on AvgBitrate and JoinTime and hit rates on BufRatio and VSF. They show that CFA outperforms other algorithms.***

caused by bulk data loading in batch mode. Note that this does not affect critical feature learning.

Iterative algorithm refinement: Some parameters (e.g., learning window size Δ_{learn}) of CFA require iterative tuning in a production environment. However, one practical challenge is that the frontend-facing part of the backend can only be updated once every couple of weeks due to code release cycles. Thus, rolling out new prediction algorithms may take several days and is a practical concern. Fortunately, the decoupling between critical feature learning and quality estimation (§4.2) means that changes to critical feature learning are confined to the backend cluster. This enables us to rapidly refine and customize the CFA algorithm.

6 Evaluation

In this section, we show that:

- CFA predicts video quality with 30% less error than competing machine learning algorithms (§6.1).
- Using CFA-based prediction, we can improve video quality significantly; e.g., 32% less BufRatio, 12% higher AvgBitrate in a pilot deployment (§6.2).
- CFA is responsive to client queries and makes predictions based on the most recent critical features and quality measurements (§6.3).

6.1 Prediction accuracy

We compare CFA with five alternative algorithms: three simple ML algorithms, Naive Bayes (NB), Decision Tree (DT), k-Nearest Neighbor (k-NN)[4], and two heuristics which predict a session's quality by the average quality of other sessions from the same ASN (ASN) or matching the last-mile connection type (LH). All algorithms use the same set of features listed in Table 1.

Ideally, we want to evaluate how accurately an algorithm can predict the quality of a given client on every choice of CDN and bitrate. However, this is infeasible since each video client is assigned to only one CDN and bitrate at any time. Thus, we can only evaluate the prediction accuracy over the observed CDN-bitrate choices, and we use the quality measured on these choices as the ground truth. That said, this approach is still useful for doing a relative comparison across different algorithms.

For AvgBitrate and JoinTime, we report *relative error*: $\frac{|p-q|}{q}$, where the q is the ground truth and p is the prediction. For BufRatio and JoinTime, which have more "step function" like effects [18], we report a slightly different measure called *hit rate*: how likely a session with good quality (i.e., BufRatio $< 5\%$, VSF=0) or bad quality is correctly identified. Figure 8 shows that for AvgBitrate and JoinTime, CFA has the lowest $\{5, 10, 50, 90\}\%$th percentiles of prediction error and lower 95%th percentiles than most algorithms. In particular, median error of CFA is about 30% lower than the best competing algorithm. In terms of BufRatio and VSF, CFA significantly outperforms other algorithms in the hit rate of bad quality sessions. The reason for hit rate of bad quality to be lower than that of good quality is that bad quality sessions are almost always less than good quality, which makes them hard to predict. Note that accurately identifying sessions that have bad quality is crucial as they have the most room for improvement.

6.2 Quality improvement

Pilot deployment: As a pilot deployment, we integrated CFA in a production system that provides a global video optimization service [20]. We deployed CFA on one major content provider and used it to optimize 150,000 sessions each day. We ran an A/B test (where each algorithm was used on a random subset of clients) to evaluate the improvement of CFA over a baseline random decision maker, which many video optimization services use by default (modulo business arrangement like price) [9].

Table 4 compares CFA with the baseline random decision maker in terms of the mean BufRatio, AvgBitrate and a simple QoE model ($QoE = -370 * BufRatio + AvgBitrate/20$), which was suggested by [33, 18]. Over all sessions in the A/B testing, CFA shows an improvement in both BufRatio (32% reduction) and AvgBitrate (12.3% increase) compared to the baseline. This shows that CFA is able to simultaneously optimize multiple (possibly conflicting) metrics. To put these numbers in context, our conversation with domain experts confirmed that these improvements are significant for content providers and can potentially translate into substantial benefits in engagement and revenues [2]. CFA's superior performance and that CFA is more automated than the custom algorithm indicate that domain experts were willing to invest time running longer pilot. Figure 9 provides more comparison and shows that CFA consistently outperforms the baseline over time and across different major cities in the US, connection types and CDNs.

	CFA	Baseline	Improvement
QoE	155.43	138.27	12.4%
BufRatio	0.0123	0.0182	32%
AvgBitrate	3200	2849	12.31%

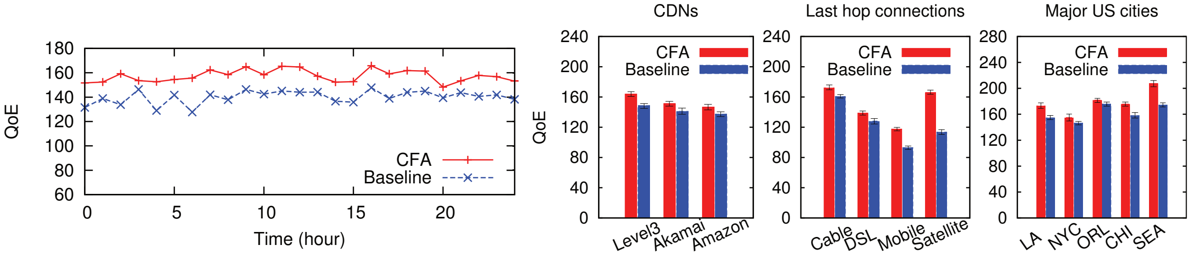
Table 4: ***Random A/B testing results of CFA vs. baseline in real-world deployment.***

Trace-driven simulation: We complement this real-world deployment with a trace-driven simulation to simultaneously compare more algorithms over more quality metrics. However, one key challenge is that it is hard to estimate the quality of a decision that was not used by a specific client in the trace.

To address this problem, we use the *counterfactual methodology* from prior work in online recommendation systems [30, 31]. Suppose we have quality measurements from a set of clients, where client c is assigned to a decision $d_{rand}(c)$ of CDN and bitrate at random. Now, we have a new hypothetical algorithm that maps client c to $d_{alg}(c)$. Then, we can evaluate the average quality of clients assigned to each decision d, $\{c|d_{alg}(c) = d\}$, by the average quality of $\{c|d_{alg}(c) = d, d_{rand}(c) = d\}$. Finally, the average quality of the new algorithm is the weighted sum of average quality of all decisions, where the weight of each decision is the fraction of sessions assigned to it. This can be proved to be an unbiased (offline) estimate of d_{alg}'s (online) performance [5].[5] For instance, if out of 1000 clients assigned to use Akamai and 500Kbps, 200 clients are assigned to this decision in the random assignment, then we can use the average quality of these 200 sessions as an unbiased estimate of the average quality of these 1000 sessions. Fortunately, our dataset includes a (randomly chosen) portion of clients with randomized decision assignments (i.e.,

[4] NB, DT, and k-NN are mplemented using a popular ML library `weka`[6].

[5] One known limitation of this analysis is that it assumes the new assignments do not affect each decision's overall performance. For instance, if we assign all sessions to one CDN, they may overload the CDN and so this CDN's quality in the random assignments is no longer useful. Since this work only focuses on controlling traffic at a small scale relative to the total load on the CDN (and our experiments are in fact performed at such a scale), this methodology is still unbiased.



(a) CFA vs. baseline by time

(b) CFA vs. baseline by spatial partitions

Figure 9: ***Results of real-world deployment. CFA outperforms the baseline random decision maker (over time and across different large cities, connection types and CDNs).***

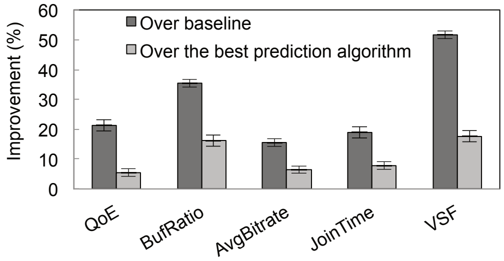


Figure 10: ***Comparison of quality improvement between CFA and strawmen.***

Stage	**Run time (mean / median)**	**Required freshness**
Critical feature learning	30.1/29.5 min	30-60 min
Quality estimation	30.7/28.5 sec	1-5 min
Query response	0.66/0.62 ms	1 ms

Table 5: ***Each stage of CFA is refreshed to meet the required freshness of its results.***

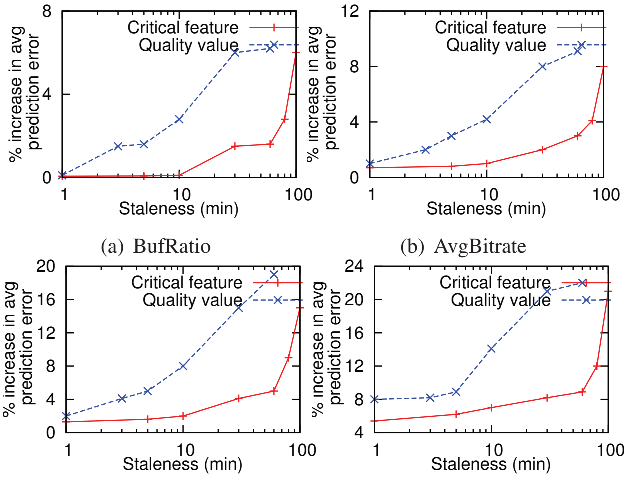
CDN and bitrate). Thus, we only report improvements for these clients.

Figure 10 uses this counterfactual methodology and compares CFA with the best alternative from §6.1 for each quality metric and the baseline random decision maker (e.g., the best alternative of AvgBitrate is k-NN). For each quality metric and prediction algorithm, the decision maker selects the CDN and bitrate that has the best predicted quality for each client. For instance, the improvement of CFA over the baseline on VSF is 52% – this means the number of sessions with start failures is 52% less than when the baseline algorithm is used. The figures show that CFA outperforms the baseline algorithm by 15%-52%. They also show that CFA outperforms the best prediction algorithms by 5%-17%.

6.3 Timeliness of prediction

Our implementation of CFA should (1) retain freshness to minimize the impact of staleness on prediction accuracy, and (2) be responsive to each prediction query.

We begin by showing how fast each stage described in §4.2 needs to be refreshed. Figure 11 shows the impact of staleness of critical features and quality values on the prediction accuracy of CFA. First, critical features learned 30-60 minutes before prediction can still achieve similar accuracy as those learned 1 minute before prediction. In contrast, quality estimation cannot be more than 10 minutes prior to when prediction is made (which corroborates the results of Figure 4(b)). Thus, critical feature learning needs to be refreshed every 30-60 minutes and quality estimation should be refreshed at least every several minutes. Finally, prediction queries need to be responded to within several milliseconds [20] (ignoring network delay between clients and servers).



(a) BufRatio

(b) AvgBitrate

(c) JoinTime

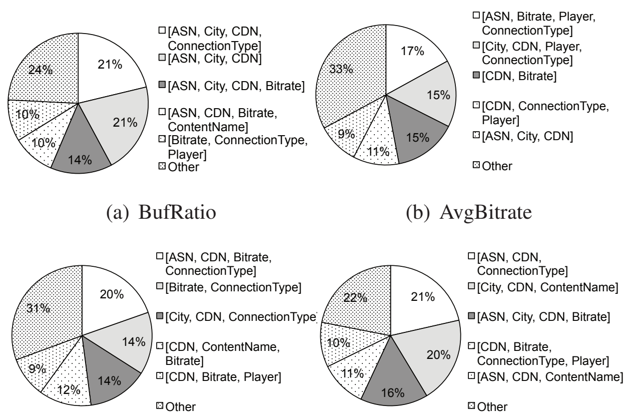
(d) VSF

Figure 11: ***Latency of critical features and quality values (x-axis) on increase in accuracy (y-axis).***

Next, we benchmark the time to run each logical stage described in §4.2. Real-time query/response runs in 4 geographically distributed data centers. Critical feature learning and quality estimation run on two clusters of 32 cores. Table 5 shows the time for running each stage and the timescale required to ensure freshness. It confirms that the implementation of CFA is sufficient to ensure the freshness of results in each stage.

7 Insights from Critical Features

In addition to the predictive power, CFA also offers insights into the "structure" of video quality in the wild. In this section, we focus on two questions: (1) What types



(a) BufRatio

(b) AvgBitrate

(c) JoinTime

(d) VSF

Figure 12: ***Analyzing the types of critical features: This shows a breakdown of the total number of sessions assigned to a specific type of critical features.***

of critical features are most common? (2) What factors have significant impact on video quality?

7.1 Types of critical features

Popular types of critical features: Figure 12 shows a breakdown of the fraction of sessions that are assigned to a specific type of critical feature set. We show this for different quality metrics. (Since we focus on a specific VoD provider, we do not consider the *Site* or *LiveOrVoD* for this analysis.) Across all quality metrics, the most popular critical features are *CDN*, *ASN* and *ConnectionType*, which means video quality is greatly impacted by network conditions at the server (*CDN*), transit network (*ASN*), and last-mile connection (*ConnectionType*).

We also see interesting patterns unique to individual metrics. *City* is among the top critical features of BufRatio. This is perhaps because network congestion usually depends on the volume of concurrent viewers in a specific region. *Bitrate* (initial bitrate) has a larger impact on AvgBitrate than on other metrics, since the videos in the dataset are mostly short content (2-5 minutes) and AvgBitrate is correlated with initial bitrate. Finally, *ContentName* has a relatively large impact on failures (VSF) but not other metrics, because VSF is sometimes due to the requested content not being ready.

Distribution of types of critical features: While the quality of about 50% of sessions is impacted by 3-4 popular types of critical features, 15% of sessions are impacted by a diverse set of more than 30 types of critical feature (not shown). This corroborates the need for expressive prediction models that handle the diverse factors affecting quality (§2.2).

7.2 Values of critical features

Next, we focus on the most prevalent *feature values* (e.g., a specific ASN or player). To this end, we define *prevalence* of a feature value by the fraction of video sessions matching this feature value that have this feature as one of their critical features; e.g., the fraction of video sessions from Boston that have *City* as one of their critical features. If a feature value has a large prevalence, then the quality of many sessions that have this feature value can be explained by this feature.

	City	*ASN*	*Player*	*ConnectionType*
BufRatio	Some major east-coast cities			Satellite, Mobile, Cable
AvgBitrate		Cellular carriers	Players with different encodings	
JoinTime		Cellular carrier		Satellite, DSL
VSF		Small ISPs		Satellite, Mobile

Table 6: ***Analysis of the most prevalent values of critical features. A empty cell implies that we found no interesting values in this combination.***

We present the values of critical features with a prevalence higher than 50% for each quality metric and only consider a subset of the features (*ASN*, *City*, *ContentName*, *ConnectionType*) that appear prominently in Figure 12. We present this analysis with two caveats. First, due to proprietary concerns, we do not present the names of the entities, but focus on their characteristics. Second, we cannot confirm some of our hypothesis as it involves other providers; as such, we intend this result to be illustrative rather than conclusive.

Table 6 presents some anecdotal examples we observed. In terms of BufRatio, we see some of the major east coast cities (e.g., Boston, Baltimore) are more likely to be critical feature values than other smaller cities. We also see both poor (Satellite, Mobile) and broadband (Cable) connection types have high prevalence on BufRatio and JoinTime. This is because poor quality sessions are bottlenecked by poor connections, while some good quality sessions are explained by their broadband connections. "Player" has a relatively large prevalence on AvgBitrate, because the content provider uses different bitrate levels for different players (Flash or iOS). Finally, in terms of VSF, some small ISPs have large prevalence. We speculate that this is because their peering relationships with major CDNs are not provisioned, so their video sessions have relatively high failure rates.

8 Related Work

Internet video optimization: There is a large literature on measuring video quality in the wild (e.g., content popularity [38, 55], quality issues [25] and server selection [51, 47]) and techniques to improve user experience (e.g., bitrate adaptation algorithms [56, 26, 23], CDN optimization and federation [32, 37, 11, 35] and cross-provider cooperation [57, 19, 24]). Our work builds on insight from this prior work. While a case for a similar vision is made in [33], our work gives a systematic and practical prediction system.

Global coordination platform: Decision making based on a global view is similar to other logically centralized control systems (e.g., [14, 33, 20, 48]). They examined the architectural issues of decoupling the control plane from the data plane, including scalability (e.g., [50, 17]), fault tolerance (e.g., [36, 54]), and use of big data systems (e.g., [20, 4]). In contrast, our work offers concrete algorithmic techniques over such a control platform [20] for video quality optimization.

Large-scale data analytics in system design: Many studies have applied data-driven techniques for performance diagnosis (e.g., [46, 15, 40]), revenue debugging (e.g., [13]), TCP throughput prediction (e.g., [22, 34]), and tuning TCP parameters (e.g., [43, 41]). Recent studies try to operate these techniques at scale [16]. While CFA shares this data-driven approach, we exploit video-specific insights to achieve scalable and accurate prediction based on a global view of quality measurements.

QoE models: Prior work has shown correlations between various video quality metrics and user engagement (e.g., users are sensitive to BufRatio [18]), and built various QoE models (e.g., [28, 45, 12, 10]. Our work focuses on improving QoE by predicting individual quality metrics, and can be combined with these QoE models.

9 Discussion

Relationship to existing ML techniques: CFA is a domain-specific prediction system that outperforms some canonical ML algorithms (§6.1). We put CFA in the context of three types of ML algorithms.

- *Multi-armed bandit* algorithms [53] find the decision with the highest reward (i.e., best CDN and bitrate) from multiple choices. They assume each decision has a fixed distribution of rewards, but the video quality of a CDN also depends on client-side features. In contrast, contextual multi-armed bandit algorithms [44] assume the best decision depends on contextual information, but they require appropriate modeling between the context and decision space, to which critical features provide one viable approach.
- *The feature selection* problem [21] seems similar to critical feature learning, but with a key difference: critical features vary across video sessions. Thus, techniques looking for features that are most important for all sessions are not directly applicable.
- *Advanced ML* algorithms today can handle highly complex models [29, 42] efficiently, so in theory the critical features could be automatically identified, albeit in an implicit manner. CFA uses existing ML models (specifically, the "variable kernel conditional density estimation" method [49]) and may be less accurate than advanced ML techniques, but CFA can predict with more recent data since it tolerates stale update on the critical features. Furthermore, CFA is less opaque since it is based on domain-specific insights about critical features (§7).

Finer grain information and selection: Currently, CFA makes predictions based on client side information only. While clients provide accurate information regarding QoE, prediction can be much more accurate if CFA were to leverage finer-grained information from other entities in the ecosystem, including servers, caches and network paths. Furthermore, CFA currently selects resources at the CDN granularity. This means CFA cannot do much if the CDN redirects the client based on its location and the servers the CDN redirects the client to are congested. However, if the client were able to specify the server to stream from, we could avoid the overloaded servers and improve quality.

10 Conclusions

Many prior research efforts posited that quality prediction could lead to improved QoE (e.g., [33, 11, 35, 10]). However, these efforts failed to provide a prescriptive solution that (a) is expressive enough to tackle the complex feature-quality relationships observed in the wild and (b) can provide near real-time quality estimates. To this end, we developed CFA, a solution based on domain-specific insights that video quality is typically determined by a subset of critical features which tend to be persistent. CFA leverages these insights to engineer an accurate algorithm that outperforms off-the-shelf machine learning approaches and lends itself to a scalable implementation that retains model freshness. Using real deployments and trace-driven analyses, we showed that CFA achieves up to 30% improvement in prediction accuracy and 12-32% improvement in QoE over alternative approaches.

Acknowledgments

This paper would not be possible without the contribution of Conviva Stuff, especially Jibin Zhan, Faisal Zakaria Siddiqi and Rui Zhang. The authors thank Siddhartha Sen for shepherding the paper and the NSDI reviewers for their feedback. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, Guavus, HP, Huawei, Intel, Microsoft, Pivotal, Samsung, Schlumberger, Splunk, State Farm, Virdata and VMware. Junchen Jiang was supported in part by NSF award CNS-1345305 and Juniper Networks Fellowship.

References

[1] Conviva inc. http://www.conviva.com/.

[2] Personal communication with aditya ganjam from conviva, who is an expert on video qoe.

[3] Quova. http://developer.quova.com/.

[4] Spark. http://spark.incubator.apache.org/.

[5] Technical note on counterfactual evaluation. https://www.cs.cmu.edu/cfe_technote.pdf.

[6] The weka manual 3.6.10. http://goo.gl/ISSY3c.

[7] Youtube and the olympics. http://goo.gl/4hgL4q.

[8] I. Sodagar. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE Multimedia*, 2011.

[9] V. K. Adhikari, Y. Guo, F. Hao, V. Hilt, , and Z.-L. Zhang. A Tale of Three CDNs: An Active Measurement Study of Hulu and Its CDNs. In *Proc. IEEE Global Internet Symposium*, 2012.

[10] V. Aggarwal, E. Halepovic, J. Pang, S. Venkataraman, and H. Yan. Prometheus: toward quality-of-experience estimation for mobile apps from passive network measurements. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, 2014.

[11] A. Balachandran, V. Sekar, A. Akella, and S. Seshan. Analyzing the potential benefits of cdn augmentation strategies for internet video workloads. In *ACM IMC*, pages 43–56, 2013.

[12] A. Balachandran, V. Sekar, A. Akella, S. Seshan, I. Stoica, and H. Zhang. Developing a predictive model of quality of experience for internet video. In *ACM SIGCOMM '13*.

[13] R. Bhagwan, R. Kumar, R. Ramjee, G. Varghese, S. Mohapatra, H. Manoharan, and P. Shah. Adtributor: revenue debugging in advertising systems. In *USENIX NSDI*, 2014.

[14] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *USENIX NSDI 2005*.

[15] D. R. Choffnes, F. E. Bustamante, and Z. Ge. Crowdsourcing service-level network event monitoring. In *ACM SIGCOMM CCR*, volume 40, pages 387–398. ACM, 2010.

[16] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[17] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed sdn controller. In *ACM HotSDN*, 2013.

[18] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. A. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. In *Proc. SIGCOMM*, 2011.

[19] B. Frank, I. Poese, Y. Lin, G. Smaragdakis, A. Feldmann, B. Maggs, J. Rake, S. Uhlig, and R. Weber. Pushing cdn-isp collaboration to the limit. *ACM SIGCOMM CCR*, 43(3), 2013.

[20] A. Ganjam, F. Siddiqi, J. Zhan, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-scale control plane for video quality optimization. In *NSDI*. USENIX, 2015.

[21] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

[22] Q. He, C. Dovrolis, and M. Ammar. On the predictability of large transfer tcp throughput. *ACM SIGCOMM CCR*, 35(4):145–156, 2005.

[23] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A buffer-based approach to rate adaptation: evidence from a large video streaming service. In *ACM SIGCOMM 2014*.

[24] J. Jiang, X. Liu, V. Sekar, I. Stoica, and H. Zhang. Eona: Experience-oriented network architecture. In *ACM HotNets*, 2014.

[25] J. Jiang, V. Sekar, I. Stoica, and H. Zhang. Shedding light on the structure of internet video quality problems in the wild. In *CoNEXT*. ACM, 2013.

[26] J. Jiang, V. Sekar, and H. Zhang. Improving Fairness, Efficiency, and Stability in HTTP-Based Adaptive Streaming with Festive . In *ACM CoNEXT 2012*.

[27] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, 2011.

[28] S. S. Krishnan and R. K. Sitaraman. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. In *IMC*, 2012.

[29] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[30] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670. ACM, 2010.

[31] L. Li, W. Chu, J. Langford, and X. Wang. Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 297–306. ACM, 2011.

[32] H. Liu, Y. Wang, Y. R. Yang, A. Tian, and H. Wang. Optimizing Cost and Performance for Content Multihoming. In *Proc. SIGCOMM*, 2012.

[33] X. Liu, F. Dobrian, H. Milner, J. Jiang, V. Sekar, I. Stoica, and H. Zhang. A case for a coordinated internet video control plane. In *ACM SIGCOMM*, pages 359–370. ACM, 2012.

[34] M. Mirza, J. Sommers, P. Barford, and X. Zhu. A machine learning approach to tcp throughput prediction. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 97–108. ACM, 2007.

[35] M. K. Mukerjee, D. Naylor, J. Jiang, D. Han, S. Seshan, and H. Zhang. Practical, real-time centralized control for cdn-based live video delivery. In *ACM SIGCOMM*, pages 311–324. ACM, 2015.

[36] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker. Cap for networks. In *ACM HotSDN*, 2013.

[37] L. Peterson and B. Davie. Framework for cdn interconnection. 2013.

[38] L. Plissonneau and E. Biersack. A longitudinal view of http video streaming performance. In *Proc. MMSys*, 2012.

[39] W. B. Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*, volume 703. John Wiley & Sons, 2007.

[40] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *USENIX NSDI*, 2011.

[41] S. Savage, N. Cardwell, and T. Anderson. The case for informed transport protocols. In *HotOS*, pages 58–63. IEEE, 1999.

[42] B. Schölkopf and A. J. Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[43] S. Seshan, M. Stemm, and R. H. Katz. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, pages 1–13, 1997.

[44] A. Slivkins. Contextual bandits with similarity information. *The Journal of Machine Learning Research*, 15(1):2533–2568, 2014.

[45] H. H. Song, Z. Ge, A. Mahimkar, J. Wang, J. Yates, Y. Zhang, A. Basso, and M. Chen. Q-score: Proactive Service Quality Assessment in a Large IPTV System. In *Proc. IMC*, 2011.

[46] M. Stemm, R. Katz, and S. Seshan. A network measurement architecture for adaptive applications. In *INFOCOM*, volume 1, pages 285–294. IEEE, 2000.

[47] A.-J. Su, D. R. Choffnes, A. Kuzmanovic, and F. E. Bustamante. Drafting behind akamai (travelocity-based detouring). *ACM SIGCOMM CCR*, 2006.

[48] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *USENIX NSDI*, 2015.

[49] G. R. Terrell and D. W. Scott. Variable kernel density estimation. *The Annals of Statistics*, pages 1236–1265, 1992.

[50] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, 2012.

[51] R. Torres, A. Finamore, J. R. Kim, M. Mellia, M. M. Munafo, and S. Rao. Dissecting Video Server Selection Strategies in the YouTube CDN. In *ICDCS*, 2011.

[52] A. Vellido, J. Martin-Guerroro, and P. Lisboa. Making machine learning models interpretable. In *Proceedings of the 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN). Bruges, Belgium*, pages 163–172, 2012.

[53] R. Weber et al. On the gittins index for multiarmed bandits. *The Annals of Applied Probability*, 2(4):1024–1033, 1992.

[54] H. Yan, D. A. Maltz, T. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: A 4d network control plane. In *NSDI*, volume 7, pages 27–27, 2007.

[55] H. Yin et al. Inside the Bird's Nest: Measurements of Large-Scale Live VoD from the 2008 Olympics. In *Proc. IMC*, 2009.

[56] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *SIGCOMM*, pages 325–338. ACM, 2015.

[57] M. Yu, W. Jiang, H. Li, and I. Stoica. Tradeoffs in cdn designs for throughput oriented traffic. In *CoNEXT*, pages 145–156. ACM, 2012.

Passive Wi-Fi: Bringing Low Power to Wi-Fi Transmissions

Bryce Kellogg[†], Vamsi Talla[†], Shyamnath Gollakota and Joshua R. Smith
University of Washington
[†]Co-primary Student Authors

Abstract – Wi-Fi has traditionally been considered a power-consuming communication system and has not been widely adopting in the sensor network and IoT space. We introduce *Passive Wi-Fi* that demonstrates for the first time that one can generate 802.11b transmissions using backscatter communication, while consuming 3–4 orders of magnitude lower power than existing Wi-Fi chipsets. Passive Wi-Fi transmissions can be decoded on any Wi-Fi device including routers, mobile phones and tablets. Building on this, we also present a network stack design that enables passive Wi-Fi transmitters to coexist with other devices in the ISM band, without incurring the power consumption of carrier sense and medium access control operations. We build prototype hardware and implement all four 802.11b bit rates on an FPGA platform. Our experimental evaluation shows that passive Wi-Fi transmissions can be decoded on off-the-shelf smartphones and Wi-Fi chipsets over distances of 30–100 feet in various line-of- sight and through-the-wall scenarios. Finally, we design a passive Wi-Fi IC that shows that 1 and 11 Mbps transmissions consume 14.5 and 59.2 μW respectively. This translates to 10000x lower power than existing Wi-Fi chipsets and 1000x lower power than Bluetooth LTE and ZigBee.

1 Introduction

Over the past few years, researchers have explored the concept of Wi-Fi backscatter [25, 38] that creates an additional narrowband data stream to ride on top of existing Wi-Fi signals. While promising, existing designs either achieve very low data rates (100s of bps) at close by distances (2-4 feet) [25] or use custom full-duplex hardware that cannot be used with any existing Wi-Fi devices [38].

In this paper, we take a different approach — instead of backscattering existing Wi-Fi signals to send an additional data stream, we use backscatter communication to *directly generate Wi-Fi transmissions* that can be decoded on any of the billions of existing devices with a Wi-Fi chipset. To this end, we introduce Passive Wi-Fi that demonstrates for the first time that one can generate 802.11b transmissions using backscatter communication, while consuming 4–5 orders of magnitude lower power than existing Wi-Fi chipsets.

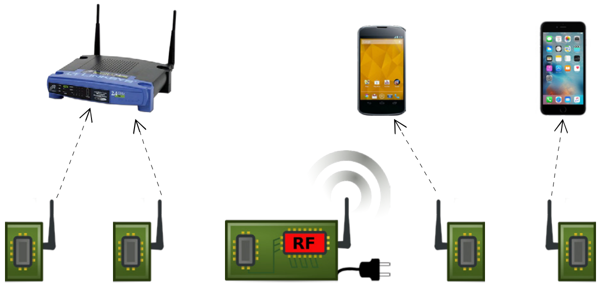


Figure 1: **Passive Wi-Fi architecture.** The passive Wi-Fi devices perform digital baseband operations like coding, while the power-consuming RF functions are delegated to a plugged-in device in the network.

We observe that while CMOS technology scaling has conventionally provided exponential benefits for the size and power consumption of digital logic systems, analog RF components, that are necessary for Wi-Fi communication, have not seen a similar power scaling. As a result, Wi-Fi transmissions on sensors and mobile devices still consume hundreds of milliwatts of power [31–33]. To get around this problem, passive Wi-Fi uses backscatter to decouple the baseband Wi-Fi digital logic from the power-consuming RF components, as shown in Fig. 1.

In our architecture, the passive Wi-Fi devices perform digital baseband operations like coding and modulation, while the power-consuming RF components such as frequency synthesizers and power amplifiers are delegated to a single plugged-in device in the network. This device provides the RF functions for all the passive Wi-Fi devices in the vicinity by transmitting a single-frequency tone. The passive Wi-Fi devices create 802.11b transmissions by reflecting or absorbing this tone using a digital switch running at baseband. Since the passive Wi-Fi devices have no analog components, they consumes less silicon area and would be smaller and cheaper than existing Wi-Fi chipsets. More importantly, their power consumption would be orders of magnitude lower since they only perform digital baseband operations. To realize this, however, we need to address three main challenges.

(a) How can Wi-Fi receivers decode in the presence of interference from the plugged-in device? The Wi-Fi receiver receives the backscattered signal in the presence of a strong interference from the tone transmitted by the plugged-in device. Traditional backscatter systems [34, 38] use a full-duplex radio to cancel this strong interfering signal, which is not possible on existing Wi-Fi devices. Our key observation is that Wi-Fi receivers are required to work even in the presence of interference in the adjacent band that is 35 dB stronger [12]. Further, as Wi-Fi and Bluetooth radios are being integrated onto a single chipset [6], Wi-Fi hardware is being designed to work in the presence of out-of-band Bluetooth interference. Thus, we set the plugged-in device to transmit its tone at a frequency that lies outside the desired Wi-Fi channel; this ensures that existing Wi-Fi chipsets can suppress the resulting out-of-band interference.

(b) How can we create 802.11b transmissions using backscatter? At a high level, we first shift the out-of-band tone from the plugged-in device to lie at the center of the desired Wi-Fi channel. We then use this shifted tone to create 802.11b transmissions. Intuitively, multiplying two sinusoidal signals can create a frequency shift.[1] Thus, by backscattering at a frequency Δf, we can shift the tone. To synthesize Wi-Fi transmissions, we leverage that 802.11b uses DSSS and CCK encoding on top of DBPSK and DQPSK modulation. The encoding operation is digital in nature and hence is achieved using digital logic. To create the phase changes required for DBPSK and DQPSK, we approximate a digital square wave as a sinusoid and modulate its phase by changing the timing of the square wave (see §2.3). Thus, passive Wi-Fi devices can fully operate in the digital domain at baseband and yet synthesize 802.11b transmissions.

(c) How do passive Wi-Fi devices share the Wi-Fi network? Traditional Wi-Fi shares the network using carrier sense. However, this requires a Wi-Fi receiver that is ON before every transmission. Since Wi-Fi receivers require power-consuming RF components such as ADCs and frequency synthesizers, this would eliminate the power savings from our design. Instead, we delegate the power-consuming task of carrier sense to the plugged-in device. At a high level, the plugged-in device performs carrier sense and signals the passive Wi-Fi device to transmit. §3 describes how such a signaling mechanism can also be used to arbitrate the channel between multiple passive Wi-Fi devices and address other link-layer issues including ACKs and retransmissions.

To show the feasibility of our design, we build prototype backscatter hardware and implement all four 802.11b bit rates on an FPGA platform. Our experimental evaluation shows that passive Wi-Fi transmissions can be decoded on off-the-shelf smartphones and Wi-Fi chipsets over distances of 30–100 feet in various line-of-sight and through-the-wall scenarios. We also design a passive Wi-Fi IC that performs 1 Mbps and 11 Mbps 802.11b transmissions and estimate the power consumption using Cadence and Synopsis toolkits [5, 19]. Our results show the 1 and 11 Mbps passive Wi-Fi transmissions consume 14.5 and 59.2 μW respectively.

Contributions. We make the following contributions:

- We demonstrate for the first time that one can generate 802.11b transmissions using backscatter communication. We present backscatter techniques that synthesize 22 MHz DSSS and CCK spread spectrum transmissions that can be decoded on existing Wi-Fi devices.
- We design a network stack for the passive Wi-Fi transmitters to coexist with other devices in the ISM band. Further, we present a detailed analytical model to understand the operational range of passive Wi-Fi transmissions in different deployment scenarios.
- We build a hardware prototype on an FPGA platform and evaluate it in various scenarios. We also design a passive Wi-Fi IC and present its power numbers.

2 Passive Wi-Fi Design

Our design has two main actors: a plugged-in device and passive Wi-Fi devices. The former contains power consuming RF components including frequency synthesizer and power amplifier and emits a single tone RF carrier. It also performs carrier sense on behalf of the passive Wi-Fi device and helps coordinate medium access control across multiple passive Wi-Fi devices. The passive Wi-Fi device backscatters the tone emitted by the plugged-in device to synthesize 802.11b transmissions that can be decoded on any device that has a Wi-Fi chipset.

In the rest of this section, we first provide a quick primer for 802.11b physical layer and backscatter communication. We then explain how the passive Wi-Fi devices generate 802.11b packets using backscatter communication. We then theoretically analyze the range of our transmissions in various deployments scenarios.

2.1 Primer for 802.11b Transmissions

802.11b is a set of Wi-Fi physical layer specifications that use spread spectrum modulation. 802.11b uses DBPSK/DQPSK at the physical layer and achieves four

[1] $2 sinft sin\Delta ft = cos(f-\Delta f)t - cos(f+\Delta f)t$.

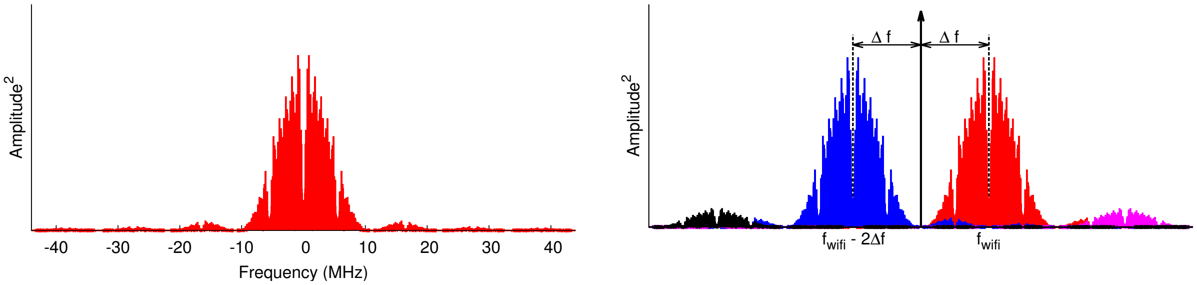


Figure 2: **Generation of Wi-Fi packets using backscatter.** The plot on the left shows the 22 MHz main lobe and the side lobes of the baseband 802.11b packet in the frequency domain. The plot on the right illustrates the backscatter operation at the passive Wi-Fi device. The two main lobes are shifted by Δf with respect to the constant tone emitted by the plugged-in device to generate the Wi-Fi packet (in red) at f_{wifi} and a mirror image (in blue) at $f_{wifi} - 2\Delta f$.

bit rates using different spreading codes. The lower two bit rates of 1 and 2 Mbps use direct-sequence spread spectrum (DSSS) while 5.5 and 11 Mbps use complementary code keying (CCK). DSSS uses a single code to spread the information over 22 MHz, while CCK uses a set of multiple code words to both encode bits and also achieve a 22 MHz spread spectrum signal. We outline how each of the 802.11b bit rates are encoded.

1 and 2 Mbps DSSS transmissions. To generate this, 802.11b first creates coded bits from the incoming data using a 11-bit barker code [39]. Specifically, 802.11b uses a single barker sequence, 10110111000, that is generated at a baseband frequency of 11 MHz to spread the spectrum over 22 MHz. To create the coded bits, 802.11b XORs each of the data bits with the barker sequence. Thus, the coded bits for a '1' data bit are 10110111000 and that for the '0' data bit are 01001000111. Each of these coded bits is encoded using DBPSK and DQPSK modulation to achieve 1 and 2 Mbps transmissions respectively. At a high level, this is achieved by setting the phase of the carrier, $sin\theta$. DBPSK modulation encodes a 0 and 1 bit by setting θ to either 0 or π, while DQPSK encodes pairs of bits by modulating the phase between 0, $\pi/2$, π and $3\pi/2$.

5.5 and 11 Mbps CCK transmissions. Instead of using a single barker code, CCK uses a set of 8-bit code words. At a high level, to generate 5.5 Mbps transmissions, the incoming data bit stream is divided into blocks of 4 bits. The first two bits are used to pick the DQPSK phase and the last two bits are used to pick a spreading code amongst four 8-bit code words. To generate 11 Mbps 802.11b transmissions, the incoming data bits are instead divided into 8 bit blocks where the first two bits determine the DQPSK phase shift and the last 6 bits are used to pick a spreading code amongst 64 8-bit code words.

To summarize, 802.11b requires generating the coded bits using either DSSS or CCK and then modulating these bits with DBPSK or DQPSK. The first operation is typically implemented in digital baseband logic while the second require changing the phase of I and Q components. Finally, we note also that since the RF energy is spread across a wide band, spread spectrum transmissions are resilient to narrowband interference both within and outside the Wi-Fi channel [39].

2.2 Backscatter Communication Primer

Unlike traditional active radio communication that requires generating RF signals, devices using backscatter communication modulate the radar cross-section of their antenna to change the reflected signal. To understand how backscatter works, consider a device that can switch the impedance of its antenna between two states. The effect of changing the antenna impedance is that the radar cross-section, i.e., the signal reflected by the antenna, also changes between the two different states. Now, given an incident signal with power $P_{incident}$, the power in the backscattered signal can be written as,

$$P_{backscatter} = P_{incident} \frac{|\Gamma_1^* - \Gamma_2^*|^2}{4} \tag{1}$$

Here Γ_1^* and Γ_2^* are the complex conjugates of the reflection coefficients corresponding to the two impedance states. Thus to maximize the power in the backscattered signal we need to maximize the difference in the power of the two impedance states which is given by $|\Delta\Gamma|^2 = \frac{|\Gamma_1^* - \Gamma_2^*|^2}{4}$. Ideally, to ensure that the power in the backscattered signal is equal to that of the incident signal, we set $|\Delta\Gamma|^2$ to 4 which can be achieved by modulating the reflection coefficients between $+1$ and -1. In practice, however, backscatter hardware deviates from this ideal behavior and incurs losses; our hardware implementation has a loss of around 1.1 dB.

2.3 802.11b using passive Wi-Fi

Generating a Wi-Fi packet using backscatter is challenging for two main reasons. First, the backscattered signal is much weaker than the tone transmitted by the plugged-in device. A Wi-Fi receiver would suffer significant in-band interference from this tone preventing it from decoding. Second, the passive Wi-Fi device has a single digital switch that toggles between two impedance states, resulting in a binary signal. It is unclear how one may generate Wi-Fi transmissions using such a binary system.

We outline how to address these challenges. We first describe the signal transmissions from the plugged-in device and then the operations at the passive Wi-Fi device that allow us to synthesize 802.11b transmissions.

Transmissions at the plugged-in device. It transmits a tone outside the desired Wi-Fi channel. Our key intuition is that Wi-Fi receivers are designed to function in the presence of out-of-band interference: 802.11b receivers are required to ensure that the sensitivity is reduced by no more than 6 dB in the presence of interference in the adjacent band that is 35 dB greater than the in-band signal [12]. Further, as Wi-Fi and Bluetooth radios are being integrated onto the same chipset [6], Wi-Fi frontends are being designed to function in the presence of out-of-band interference from Bluetooth devices. Since the tone from the plugged-in device is narrower in bandwidth than Bluetooth, this would further help suppress the tone if it is outside the desired Wi-Fi channel.

We note however that excessive out-of-band interference, which occurs when the Wi-Fi receiver is right next to the plugged-in device, can saturate and/or compress the RF front end resulting in significant degradation of Wi-Fi performance. This is called the input 1 dB compression point which is around 0 dBm for commercial Wi-Fi devices [13]. Passive Wi-Fi inherently avoids this issue by ensuring that the Wi-Fi receiver (e.g., smartphone or router) is not next to the plugged-in device.

Backscatter operations at passive Wi-Fi devices. At a high level, the passive Wi-Fi operations can be described as first shifting the out-of-band tone transmitted from the plugged-in device to lie at the center of the desired Wi-Fi channel. We then use this shifted tone to create 802.11b transmissions. To do this, we leverage three key facts: (1) From basic trigonometry, $2 sinft sin\Delta ft = cos(f-\Delta f)t - cos(f+\Delta f)t$. Thus, multiplying two sinusoidal signals can create a frequency shift. (2) Modulating the radar cross section of an antenna effectively multiplies the incoming signal by the modulated signal. Thus, modulating the antenna at a frequency Δf would create a frequency shift in the incoming signal. (3) All bit rates in 802.11b are differentially phase modulated using DBPSK or DQPSK.

Step 1. Shifting the tone from the plugged-in device using backscatter. Say the plugged-in device sends the tone $sin 2\pi(f_{wifi} - \Delta f)t$ outside the Wi-Fi channel. Passive Wi-Fi devices use a square wave at a frequency of Δf to shift the tone to the center of the Wi-Fi channel. From Fourier analysis, a square wave can be written as,

$$Square(\Delta ft) = \frac{4}{\pi} \sum_{n=1,3,5,..}^{\infty} \frac{1}{n} sin(2\pi n \Delta ft)$$

Here the first harmonic is a sinusoidal signal at the desired frequency Δf. Note that the power in each of these harmonic scales as $\frac{1}{n^2}$. So the third and the fifth harmonic are around 9.5 dB and 14 dB lower than the first harmonic. Thus, we can approximate a square wave as just the sinusoidal signal, $\frac{4}{\pi} sin(2\pi\Delta ft)$. Since modulating the radar cross section of an antenna effectively multiplies the incoming signal by the modulated signal, the backscatter signal can be approximated as $sin 2\pi(f_{wifi} - \Delta f)t sin 2\pi\Delta ft$. So we have used backscatter to effectively creates two tones, one centered at f_{wifi} and the other at $f_{wifi} - 2\Delta f$; the first tone is at the center of the desired Wi-Fi channel.

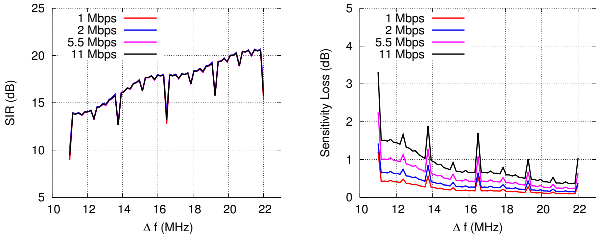
Step 2. Synthesizing 802.11b transmissions using backscatter. Now that we have a tone centered at the Wi-Fi channel, the next step is to create 802.11b transmissions using backscatter. 802.11b uses DSSS and CCK encoding which are both digital operations and hence can be performed using digital logic at the passive Wi-Fi device. So the question that remains is: how do we generate DBPSK and DQPSK using just a square wave created at a frequency Δf by the backscatter switch?

Passive Wi-Fi does this by noting that DBPSK and DQPSK use a sine wave with four distinct phases: $0, \pi/2, \pi, 3\pi/2$. Since the square wave generated by our digital switch can be approximated as a sine wave, we can generate the required four phases by changing the timing of our square wave. Specifically, shifting the square wave by half of a symbol time, effectively creates a phase change of π. Phase changes of $\pi/2$ and $3\pi/2$ can be achieved by shifting the square wave by one-fourth and three-fourth of a symbol time. Thus, passive Wi-Fi devices can fully operate in the digital domain while run at a baseband frequency of a few tens of MHz and synthesize 802.11b transmissions using backscatter.

We note the following properties of our design.

- In addition to creating a 802.11b transmission centered at f_{wifi}, as shown in Fig. 2, our backscatter mechanism also creates a mirror copy centered at $f_{wifi} - 2\Delta f$ on the other side of the tone. Thus, we use twice the bandwidth of a traditional 802.11b transmission. This is the tradeoff we make to achieve orders of magnitude lower power consumption. We note that such a tradeoff is common in 802.11n systems which use channel bonding of adjacent Wi-Fi channels to double the throughput.

- 802.11b transmissions have side lobes (Fig. 2); the side lobes of the mirror copy creates interference for the desired Wi-Fi signal. We plot the signal to interference



(a) Signal to Interference ratio *(b) Sensitivity Loss*

Figure 3: **SIR and loss in receiver sensitivity.** The plot shows the effect of different Δf's on the quality and the sensitivity of the synthesized Wi-Fi packets.

ratio for different frequency shifts, Δf, at the passive Wi-Fi device. Fig. 3(a) plots the results for all four 802.11b bit rates and shows that the interference from the side lobes of the mirror copy reduces as Δf increases. This is because, as Δf increases, the mirror copies are further separated in frequency, resulting in lower interference.

- An effect of this interference, however, is that it adds additional noise to the Wi-Fi signal, reducing the noise sensitivity at which each of the 802.11b bit rates can be decoded. Fig. 3(b) shows the loss in sensitivity for the four 802.11b bit rates, as a function of the frequency offset, Δf. The plots show that the sensitivity loss is slightly larger for higher 802.11b bit rates. This is because higher bit rates require a cleaner signal to successfully be decoded. Our system sets Δf to 12.375 MHz, where the sensitivity loss is less than 2 dB across all 802.11b bit rates. This also ensures that the passive Wi-Fi transmissions only occupy two adjacent Wi-Fi channels. Note that Wi-Fi applies filters to remove the interfering side lobes. Our implementation however does not do this.

2.4 Analyzing Passive Wi-Fi's Range

In passive Wi-Fi, the communication range depends on two parameters: the distance between the plugged-in device and the passive Wi-Fi transmitter and the distance between the passive Wi-Fi transmitter and the Wi-Fi receiver. Specifically, the signal strength at the receiver, P_r, can be modeled using Friis path loss [34] as follows,

$$P_r = \left(\frac{P_t G_t}{4\pi d_1^2}\right)\left(\frac{\lambda^2 G_{passive}^2}{4\pi}\frac{|\Delta\Gamma|^2}{4}\alpha_{wifi}\right)\left(\frac{1}{4\pi d_2^2}\frac{\lambda^2 G_r}{4\pi}\right)$$

This equation has three key parts: the term in first parenthesis models signal propagation from the plugged-in device, with an output power P_t and an antenna gain G_t, to a passive Wi-Fi transmitter at a distance d_1 away. The third term, similarly, models the signal propagation from the passive Wi-Fi transmitter to a Wi-Fi receiver with an antenna gain G_r and at a distance d_2 away. Here, λ is the wavelength of the RF signal been transmitted. Finally, the middle parenthesis models the fraction of incident signal from the plugged-in device that is backscattered by a passive Wi-Fi transmitter with an antenna gain $G_{passive}$. $|\Delta\Gamma|^2$ is the backscatter coefficient which is a measure of the efficiency with which passive Wi-Fi can generate backscatter signals. As described in §2.2, this is 1.1 dB in our hardware. Finally, α_{wifi} models the loss in energy due to synthesis of Wi-Fi signals using backscatter. This is around 4.4 dB and includes half the power lost in the mirror copy generated by backscatter and the losses due to the side lobes as described in §2.3.

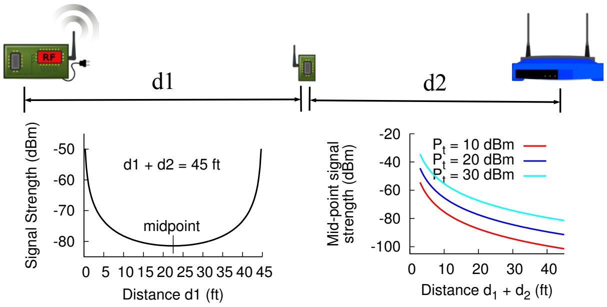


Figure 4: **Passive Wi-Fi's analytical received signal strength.** The passive Wi-Fi device moves along the line connecting the Wi-Fi router and plugged-in device.

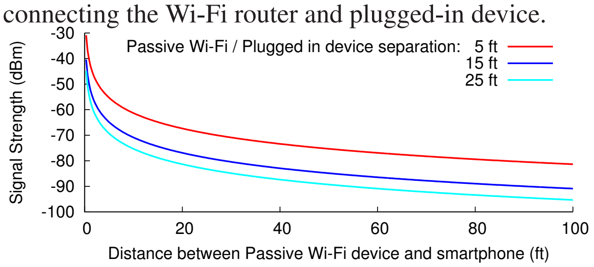
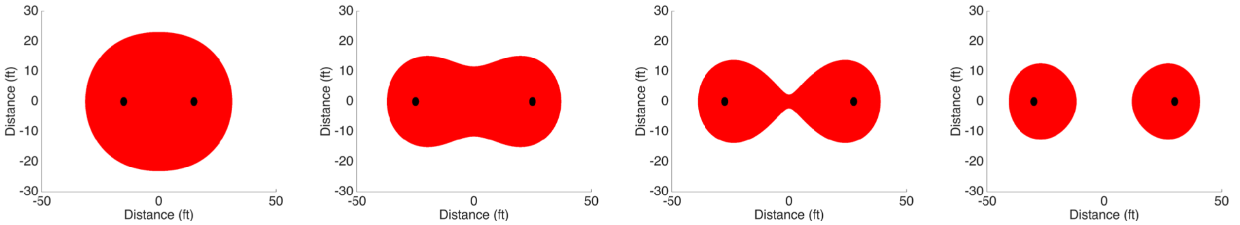


Figure 5: **Signal strength versus distance between passive Wi-Fi transmitter and Wi-Fi receiver.**

To gain a better intuition, consider the scenario in Fig. 4 where we place the plugged-in device and the Wi-Fi receiver separated by 45 feet. We move the passive Wi-Fi transmitter between these devices, along the line connecting them. We set P_t, G_t, G_r and $G_{passive}$ to 30 dBm, 6 dBi, 0 dBi, and 2 dBi respectively. Fig. 4 shows the received signal strength, P_r, as we move the passive Wi-Fi transmitter between the plugged-in device and the Wi-Fi receiver. The plots show two key points.

(1) The received signal increases as the passive Wi-Fi transmitter gets close to either the Wi-Fi receiver or the plugged-in device. This is because, maximizing the signal strength requires minimizing the product $d_1 d_2$, which is achieved either by reducing the distance d_1 or d_2.

(2) The mid-point between the plugged-in device and Wi-Fi receiver has the lowest strength. Fig. 4 shows this mid-point signal strength, as we change the distance between the plugged-in device and Wi-Fi receiver. The plot shows that this decreases with distance between the plugged-in device and the Wi-Fi receiver. As expected, it increases with plugged-in device's transmit power (P_t).



(a) 30 ft Separation (b) 50 ft Separation (c) 55 ft Separation (d) 60 ft Separation

Figure 6: **Theoretical coverage maps for different distances between the plugged-in device and the Wi-Fi router.** The black dots denote the positions for these devices. The red region represents points in the 2D space where a passive Wi-Fi transmitter can be located, while ensuring that the signal from it to the Wi-Fi router is at least -85 dBm.

2.4.1 Understanding Deployment Scenarios

1. I want to deploy passive Wi-Fi devices in my home. Where do I place the plugged-in device so as to maximize their range? Fig. 5 shows the theoretical signal strength at the Wi-Fi receiver as a function of its distance from the passive Wi-Fi transmitter. We show the results for different distances between the passive Wi-Fi transmitter and the plugged-in device. We set G_t, G_r, $G_{passive}$, P_t to 6 dBi, 0 dBi, 2 dBi, and 30 dBm respectively. The plot shows that, in general, as the distance between the passive Wi-Fi transmitter and Wi-Fi receiver increases, the received signal strength reduces. More importantly, as the distance between the passive Wi-Fi transmitter and plugged-in device decreases, the coverage range increases. This is because, from our analysis, the signal strength can be increased either by reducing the distance between the passive Wi-Fi transmitter and the plugged-in device or that between the passive Wi-Fi transmitter and the Wi-Fi receiver. Since our goal is to maximize range, we should reduce the distance between the passive Wi-Fi transmitter and the plugged-in device. In the presence of multiple passive Wi-Fi devices, this would translate to minimizing the worst-case distance between the plugged-in device and all passive Wi-Fi transmitters.

2. Where do I place my Wi-Fi router and the plugged-in device, so that I can have passive Wi-Fi devices work from anywhere in my home? Fig. 6 shows the 2D coverage maps for different distances between the plugged-in device and the Wi-Fi router. The red region represent points in the 2D space where a passive Wi-Fi transmitter can be located, while ensuring that the signal from it to the Wi-Fi router is at least -85 dBm. These maps show that the coverage area is a union of two circles centered each at the Wi-Fi router and the plugged-in device. So, as a general rule of thumb, it is better to deploy the plugged-in device and the Wi-Fi router at either ends of the coverage area. Note however that at very large distances between the plugged-in device and Wi-Fi router (Figs. 6 (c) and (d)), we end up getting two islands of coverage. Such large distance deployments are suitable only when the passive Wi-Fi transmitters are going to be close to either the plugged-in device or the Wi-Fi router.

0 : 9	10	11 : 12	13 : 15
Device ID	Ack	Rate	Check Bits

Figure 7: **Structure of the signaling packet.**

3 Passive Wi-Fi Network Stack Design

We first describe how passive Wi-Fi devices share the ISM band. We then address the issue of ACKs and retransmissions and finally, present our protocol to associate passive Wi-Fi devices with the network.

3.1 Sharing the ISM band

Wi-Fi uses carrier sense to share the ISM band. This however requires a Wi-Fi receiver that is ON before every transmission. Since Wi-Fi receivers require power-consuming RF components like LNA, frequency synthesizers, mixers and ADCs, this would eliminate the power savings from our design. Instead, we delegate the task of carrier sense to the plugged-in device, which also arbitrates access between multiple passive Wi-Fi devices.

We illustrate this with an example. Say a passive Wi-Fi transmitter wants to sent a packet on channel 6 and the plugged-in device transmits its tone between Wi-Fi channels 1 and 6. Before any of the above transmissions happen, the plugged-in device first uses carrier sense to ensure that there are no ongoing transmissions on any the frequencies including and in between channel 1 and 6.

Once the channels are found free, the plugged-in device sends a packet signaling a specific passive Wi-Fi device to transmit. This signal is sent and decoded using the ultra-low power receiver described in §3.1.1. The packet starts with an ID unique to each passive Wi-Fi device (see Fig. 7). When the passive Wi-Fi device detects its ID, it transmits within a SIFS duration at the end of the signaling packet. The signaling packet is sent at the center of channel 1 and 6 as well as in between them. This prevents other devices in the ISM band from capturing the channel before the passive Wi-Fi device gets to transmit. The packet has 16 bits and adds a fixed overhead of 100 μs for every passive Wi-Fi transmission.

The above description assumes that the plugged-in device knows when to send the signaling packet to each of the passive Wi-Fi devices in the network. To see how this can be achieved let us focus on our target IoT applications. A device sending out beacons is configured to send them at a fixed rate. Temperature sensors, microphones and Wi-Fi cameras (e.g., Dropcam [8]) have a fixed rate at which they generate data. Similarly, motion sensors have an upper bound on the delay they can tolerate. The passive Wi-Fi devices convey this information to the plugged-in device during association (and can update it later using the protocol in §3.3). This information is used by the plugged-in device to signal each passive Wi-Fi device in accordance to its desired update rate.

3.1.1 Ultra-low power receiver design

We encode bits using ON-OFF keying. We use a passive energy detector with analog components and a comparator to distinguish between the presence and absence of energy. Our design is the same as that used in our prior work [25, 26] and we skip it for brevity. We implement the receiver using off-the-shelf components and it consumes 18 μW, while achieving a bit rate of 160 kbps.

3.2 ACKs and Rate Adaptation

ACKs and retransmissions. The plugged-in device listens to the ACKs and conveys this information back to the passive Wi-Fi sensor. Specifically, if the ACK is successfully decoded at the plugged-in device, it sets the ACK bit in the signaling packet shown in Fig. 7 to 1 and sends it to the passive Wi-Fi sensor, by piggybacking it during the next period when the sensor is scheduled to transmit. If the ACK is not received at the plugged-in device, it immediately performs carrier sense and sends a signaling packet with the ACK bit set to 0. When the passive Wi-Fi sensor receives this, it retransmits its sensor value. In our implementation, the plugged-in device detects an ACK by detecting energy for a ACK duration at the end of the passive Wi-Fi transmission.

Rate adaptation. Wi-Fi bit rate adaptation algorithms typically use packet loss as a proxy to adapt the transmitter bit rate. In our design, we delegate this function to the plugged-in device. Specifically, the plugged-in device estimates the packet loss rate for each of its associated passive Wi-Fi devices by computing the fraction of successfully acknowledged packets. It then estimates the best 802.11b bit rate and encodes this information in the bit rate field of the signaling packet. Since the plugged-in device knows the bit rate as well as the packet length (from association as described in §3.3), it knows how long the transmissions from each of its passive Wi-Fi devices would occupy on the wireless medium. Thus, it stops transmitting its tone at the end of the passive Wi-Fi transmission and listens for the corresponding ACKs.

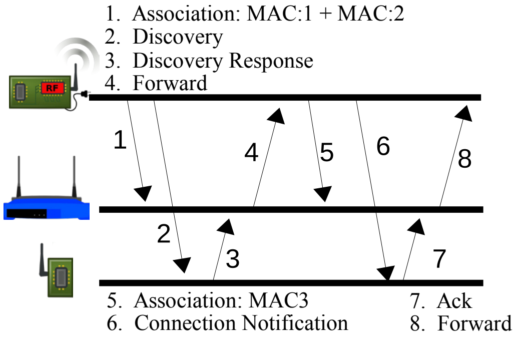


Figure 8: **Passive Wi-Fi association procedure.**

3.3 Network Association

Finally, we describe how the passive Wi-Fi transmitters associate with the plugged-in device as well as with the Wi-Fi router in the network. The key challenge is that since the plugged-in device does not have a full-duplex radio (the lack of which is desirable to make it practical and keep it low cost), there is no direct communication channel from the passive Wi-Fi device to the plugged-in device. Instead, as shown in Fig. 8, the plugged-in device associates with the Wi-Fi router with two MAC address (MAC:1 and MAC:2). The plugged-in device then broadcasts a discovery packet using ON-OFF keying modulation that contains these two MAC addresses and starts with a broadcast ID. The new passive Wi-Fi device then transmits a Wi-Fi packet with the source and destination addresses set to MAC:2 and MAC:1; this packet gets routed through the Wi-Fi router to the plugged-in device. The packet payload includes the sensor update rate, packet length, supported bit rates and its MAC address, MAC:3. The plugged-in device spoofs MAC:3 and associates it with the Wi-Fi router. It then picks a unique ID and sends it to the passive Wi-Fi device along with other Wi-Fi network credentials. Finally, the passive Wi-Fi device responds with a Wi-Fi packet with the source and destination addresses set to MAC:3 and MAC:1; this packet gets routed through the Wi-Fi router and confirms association at the plugged-in device.

After association, the passive Wi-Fi transmitter can send Wi-Fi packets to the plugged-in device through the router, and change its parameters including update rate and packet length. Note that the credentials for the spoofed MAC addresses could be sent securely using a manufacturer set secret key shared between the passive Wi-Fi devices and the plugged-in devices.

4 Hardware Implementation

We first describe our implementation of passive Wi-Fi using off-the shelf components on an FPGA platform. We use this to characterize passive Wi-Fi in various deployment scenarios. We then present our IC design

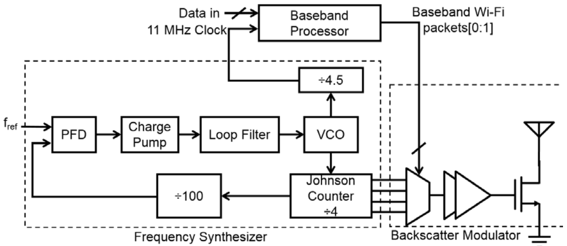


Figure 9: **Passive Wi-Fi's IC architecture.** The frequency synthesizer generates baseband clock.

which we use to quantify our power consumption.

Off-the-shelf implementation. We implement a passive Wi-Fi prototype using off-the-shelf components for backscatter and an FPGA for digital processing. The backscatter modulator consisted of HMC190BMS8 SPDT RF switch network on a 2-layer Rodgers 4350 substrate [10]. The switch was designed to modulate between open and closed impedance states and had a 1.1 dB loss. All the required baseband processing including data scrambling, header generation, DSSS/CCK encoding, CRC computation and DBPSK/DQPSK modulation were written in Verilog. The Verilog code was synthesized and programmed on a DE1 Cyclone II FPGA development board by Altera [2]. We implement four shifts of 12.375, 16.5, 22 and 44 MHz. The digital output of the FPGA was connected to the backscatter switch to generate the Wi-Fi packets. A 2 dBi omnidirectional antenna was used on the passive Wi-Fi device. The plugged-in device was set to transmit at an EIRP of 30 dBm.

Integrated circuit implementation. CMOS technology scaling has enabled the exponential scaling in power and area for integrated circuits. Wi-Fi chipsets have tried to leverage scaling but with limited success due to the need for power hungry analog components that do not scale in power and size with CMOS technology. However, baseband Wi-Fi operations are implemented in the digital domain and tend to scale very well with CMOS. For context, Atheros's AR6003 [4] and AR9462 [17] chipsets that were released in 2009 and 2012 use 65 nm CMOS and 55 nm CMOS node implementations respectively. For passive Wi-Fi device's integrated circuit implementation, we chose the 65 nm LP CMOS node by TSMC, which gives us power savings of baseband processing and ensures a fair comparison with current industry standards. The IC architecture of the passive Wi-Fi device is shown in Fig. 9 and has three main components:

Baseband frequency synthesizer. It generates the 11 MHz clock required for baseband processing as well as four phases at 12.375 MHz offsets required for DBPSK and DQPSK. We phase synchronize the 11 MHz and 12.375 MHz clocks to avoid glitches during phase modulation. We used an integer N charge pump and ring oscillator-based PLL to generate 49.5 MHz clock from a 12.375 kHz reference. The 49.5 MHz clock is fed to a quadrature Johnson counter to generate the four phases with the required timing offsets (corresponding to 0, $\frac{\pi}{2}$, π and $\frac{3\pi}{2}$ phases). The same 49.5 MHz carrier is divided by 4.5 to generate the 11 MHz baseband clock.

Table 1: **Passive Wi-Fi's IC Power Consumption**

	1 Mbps	**11 Mbps**
Baseband Frequency Synthesizer	5.6 μW	5.6 μW
Baseband Processor	5.0 μW	48 μW
Backscatter Modulator	3.9 μW	5.6 μW
Total Power	14.5 μW	59.2 μW

Baseband processor. It takes the payload bits as input and generates baseband 802.11b Wi-Fi packet. We used the Verilog code that was verified on the FPGA and use the Design Compiler by Synopsis to generate the transistor level implementation of the baseband processor [19].

Backscatter modulator. It mixes the baseband data to generate DBPSK and DQPSK and drives the switch to backscatter the incident tone signal. The baseband data are the select inputs to a 2-bit multiplexer which switches between the four phases of the 12.375 MHz clock to generate the phase modulated data. The multiplexer output is buffered and used to drive the RF switch, which toggles the antenna between open and short impedance state.

Table 1 shows the power consumption of our design at 1 Mbps and 11 Mbps which was computed using the Cadence spectre and Synopsis Design Complier toolkits [5, 19]. Passive Wi-Fi's IC implementation for 1 Mbps and 11 Mbps consumes a total of 14.5 and 59.2 μW of power respectively. The digital frequency synthesizer is clocked for DQPSK and consumes a fixed power for all data rates. The power consumption of the baseband processor that generates the 802.11b packets scales with the data rate and consumes 30% and 80% of total power for 1 and 11 Mbps respectively.

5 Evaluation

5.1 Physical Layer Performance

We first evaluate the range and then the effect of the frequency shift used in our system. Finally, we present results for all four 802.11b bit rates.

5.1.1 RSSI in Line-of-sight scenarios

We run experiments in two line-of-sight scenarios.

Deployment scenario 1. We fix the distance between the passive Wi-Fi device and the plugged-in device. We then

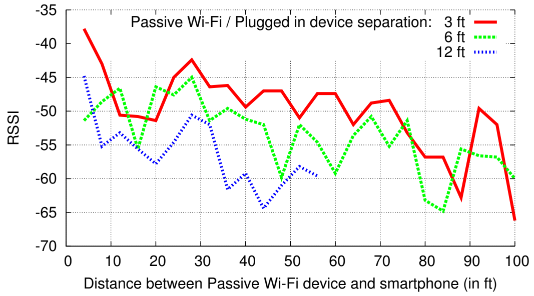


Figure 10: **RSSI in deployment scenario 1.** We move the phone away from the passive Wi-Fi device.

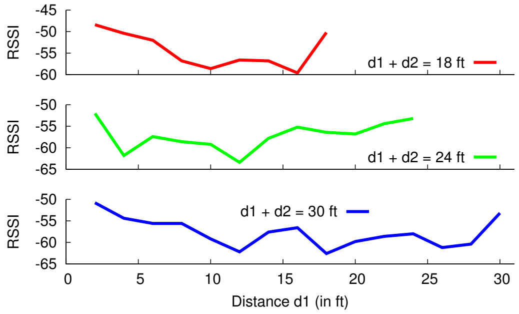
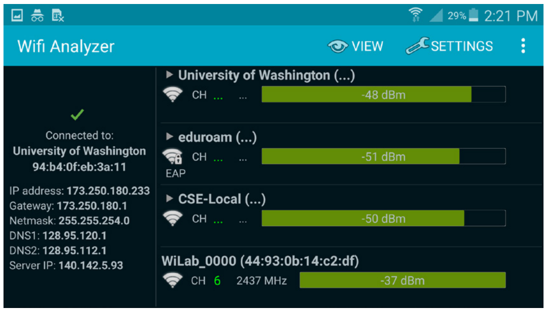


Figure 11: **RSSI in deployment scenario 2.** d_1 (d_2) is the distance between the passive Wi-Fi and plugged-in device (Wi-Fi receiver). The passive Wi-Fi device moves alone the line joining the other two devices.

move the Wi-Fi receiver away from the passive Wi-Fi device and measure the RSSI of the passive Wi-Fi transmissions as seen by the receiver. We run the experiments in the CSE atrium where the maximum distance possible when the passive Wi-Fi device and Wi-Fi receiver were placed on either end was around 100 feet. In our experiments, we set the passive Wi-Fi device to generate 802.11b beacon packets at 1 Mbps. These packets have a payload of 68 bytes where the SSID is set to *WiLab_0000* and are transmitted every 15 ms. We set the plugged-in device to transmit its tone 12.375 MHz from the center of Wi-Fi channel 1 between channel 1 and 6. We use an HTC One (M7) phone as our Wi-Fi receiver. Since the passive Wi-Fi device is transmitting Wi-Fi beacons, it appears as a Wi-Fi AP at the smartphone. To measure the RSSI values of these packets, we use a third party Android app called Wifi Analyzer [3] that provides the RSSI value as shown in Fig. 12.

In each experiment, we hold the smartphone in our hand and measure the reported RSSI values as we walk away from the passive Wi-Fi device. The measurements are taken at increments of 4 feet. Fig. 10 plots the results for three different values of the distance between the passive Wi-Fi transmitter and the plugged-in device. The x-axis plots the distance between the passive Wi-Fi transmitter and the Wi-Fi receiver while the y-axis plots the reported RSSI values. The plots show that as expected, the RSSI values reduce as the phone moves away from the passive Wi-Fi device. Further, as predicted by our analysis in §2.4, the range of our passive Wi-Fi transmissions reduce with the distance between the passive Wi-Fi transmitter and the plugged-in device. When the separation between the passive Wi-Fi transmitter and the plugged-in device is 3 or 6 feet, the range of the passive Wi-Fi transmissions spans the entire length of the CSE atrium. The range is around 55 feet when this separation is 12 feet. This reduced range is due to a combination of multipath and a weak backscatter signal.



Connected to:
University of Washington
94:b4:0f:eb:3a:11

IP address: 173.250.180.233
Gateway: 173.250.180.1
Netmask: 255.255.254.0
DNS1: 128.95.120.1
DNS2: 128.95.112.1
Server IP: 140.142.5.93

▶ University of Washington (...)
CH ... ... -48 dBm

▶ eduroam (...)
CH ... ... -51 dBm
EAP

▶ CSE-Local (...)
CH ... ... -50 dBm

WiLab_0000 (44:93:0b:14:c2:df)
CH 6 2437 MHz -37 dBm

Figure 12: **Snapshot of the Wi-Fi analyzer app.** *WiLab_0000* corresponds to passive Wi-Fi beacons.

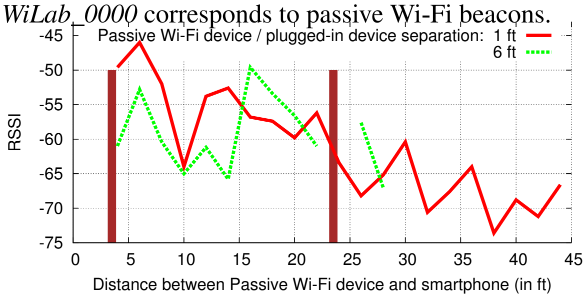


Figure 13: **RSSI in deployment scenario 1 in the presence of walls.** The brown blocks show the wall positions.

Deployment scenario 2. Next we place the plugged-in device and the Wi-Fi receiver at a distance $d_1 + d_2$. We move the passive Wi-Fi transmitter along the line connecting these two devices. As above the passive Wi-Fi transmitter is set to generate 802.11b beacon packets at 1 Mbps and the plugged-in device transmits its tone at 12.375 MHz from the center of Wi-Fi channel 1. We collect the RSSI values from a HTC One (M7). Fig. 11 plots the results for three different values of the distance between the plugged-in device and the Wi-Fi receiver ($d_1 + d_2$). Each point on the x-axis denotes the distance between the passive Wi-Fi device and the plugged-in device (d_1). The plots show that the RSSI values are the highest when the passive Wi-Fi transmitter is either close to the Wi-Fi receiver or the plugged-in device. Further, the RSSI values are lower at the mid point between the two devices, confirming our theoretical analysis.

5.1.2 RSSI in Through-the-Wall Scenarios

We rerun experiments in the above deployment scenarios but now in the presence of walls. In the first deployment,

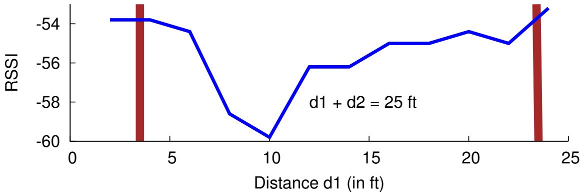


Figure 14: **RSSI in deployment scenario 2 in the presence of walls.** The brown blocks denote the walls. d_1 (d_2) is the distance between the passive Wi-Fi device and plugged-in device (Wi-Fi receiver).

we place the passive Wi-Fi device and the plugged-in devices at distances of 1 and 6 feet from each other. As the Wi-Fi receiver moves away from the passive Wi-Fi device, it is separated by multiple double sheet-rock (plus insulation) walls with a thickness of approximately 5.7 inches. As before, we use an HTC One (M7) phone as our Wi-Fi receiver and set the plugged-in device to transmit with a 12.375 MHz frequency offset from channel 1. The passive Wi-Fi device periodically transmits Wi-Fi beacons at 1 Mbps and we measure the RSSI values as reported by the Wi-Fi receiver. Fig. 13 shows that the range is now around 28 feet when the distance between the passive Wi-Fi device and the plugged-in device is 6 feet. This is expected because the signals get attenuated by two walls before arriving at the Wi-Fi receiver.

In the second deployment, we fix the location of the plugged-in device in the first room and place the Wi-Fi receiver in the third room at a distance of 25 feet. We then move the passive Wi-Fi device along the line connecting the above two devices and measure the RSSI reported by the Wi-Fi receiver. Fig. 14 plots the RSSI results and show that they follow a similar trend as before and work even in the presence of attenuation from walls.

5.1.3 Effect of different frequency shifts

We evaluate how different frequency shift values effect passive Wi-Fi performance. To do this, we place the passive Wi-Fi transmitter and plugged-in device 6 feet from each other. We move a Wi-Fi receiver away from the passive Wi-Fi device in a 50 foot long space. The passive Wi-Fi device transmits 1 Mbps Wi-Fi packets with a payload of 512 bytes on channel 1. We use the Intel 5350 chipset as a Wi-Fi receiver which runs tshark to log all the packets that are successfully decoded by it. The passive Wi-Fi transmitter consecutively transmits 200 unique sequence numbers in a loop using which we compute the packet error rate at the Wi-Fi receiver. We repeat these experiments for three different shifts.

Fig. 15 plots the PER at the Wi-Fi receiver as a function of distance between the passive Wi-Fi transmitter and the Wi-Fi receiver. The figure shows that the PER is consistently around 20% when we use frequency shifts of 44 and 16.5 MHz. For comparison, we measured the PER for a conventional Wi-Fi transmitter placed 10 feet away and observed similar PER values. The interesting observation however is that when the shift is 12.375 MHz, we see a large variation in the PER as the location of the Wi-Fi receiver changes. This is because of two related reasons. First, when the shift is small, the tone from the plugged-in device is very close to the desired Wi-Fi channel. Second, because of multipath, different locations see different signal strength differences between the passive Wi-Fi device and the out-of-band interference from the plugged-in device. When the shift is small, this out-of-band interference can still be significant in certain locations to create losses. We note that while a 44 MHz shift is too high to be within the ISM band, a 16.5 MHz shift has PERs that are stable across locations and yet is small to be within the ISM band while generating packets on all Wi-Fi channels.

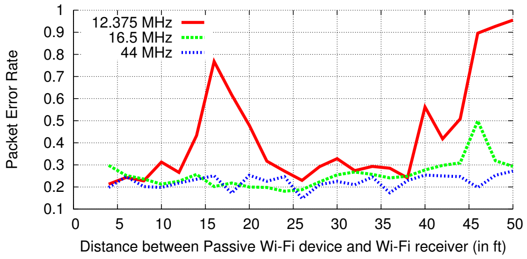


Figure 15: **Effect of different frequency shifts.** The PERs are very stable with 16.5 MHz and 44 MHz offsets.

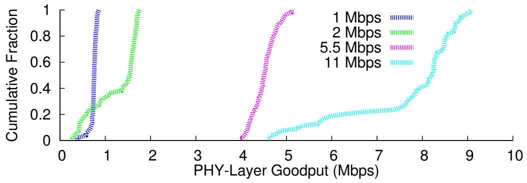
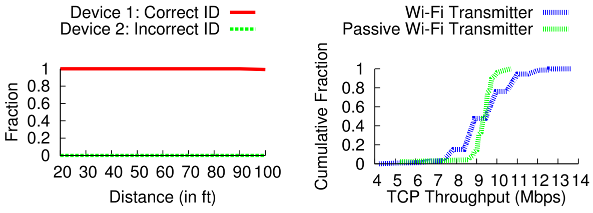


Figure 16: **All 802.11b bit rates.** Our design can generate 802.11b transmissions across all four bit rates.

5.1.4 Higher 802.11b bit rates

Finally, we show that passive Wi-Fi can generate all 802.11b bit rates. We separate the passive Wi-Fi and plugged-in device by 6 feet. We change the Wi-Fi receiver location to five spots in a 15×24 ft room. The plugged-in device is set to use a 12.375 MHz offset. For each Wi-Fi receiver location, the passive Wi-Fi device transmits 802.11b packets at 1, 2, 5.5 and 11 Mbps. For each bit rate, the passive Wi-Fi device sends 200 packets with a 512 byte payload with different sequence numbers. The Wi-Fi receiver (Intel 5350) is configured to compute the effective PHY goodput achieved by multiplying the transmitted Wi-Fi bit rate with the fraction of packets that are decoded. Fig. 16 plots a CDF of the PHY-layer goodput across the five locations demonstrating that we can generate all four 802.11b bit rates.



(a) ID Detection *(b) Wi-Fi Coexistence*

Figure 17: **Passive Wi-Fi network performance.**

5.2 Passive Wi-Fi Network Performance

As described in §3.1 to coexist in the ISM band, the plugged-in device first performs carrier sense and then signals the passive Wi-Fi device to transmit. In this section, we first evaluate how well the signaling mechanism works. We then describe how our overall carrier sense mechanism works in the presence of other Wi-Fi devices.

5.2.1 Evaluating the signaling mechanism

The plugged-in device transmits a packet with a 10-bit ID that is unique to each passive Wi-Fi device. We evaluate two aspects: (1) the probability with which the signal from the plugged-in device trigger transmissions from the correct passive Wi-Fi device and (2) the probability that it would trigger the wrong passive Wi-Fi device. To evaluate this we consider the worst-case scenario: two devices that have IDs that differ by just one bit. We set the plugged-in device to transmit the signaling packet with the ID of the first device. We move the two passive Wi-Fi devices away from the plugged-in device. At each distance value, the plugged-in device is configured to transmit the signaling packet for a total of 1890 times. The passive Wi-Fi devices use an envelope detector to correlate for their specific ID. We compute the fraction of the 1890 signaling packets that are decoded and match the ID of the passive Wi-Fi device. We run these experiments in the UW CSE atrium for increasing distances from the plugged-in device. Fig. 17(a) show the fraction of signaling packets that match the ID of the two passive Wi-Fi devices as a function of the distance from the plugged-in device. The plot shows that neither device incorrectly decodes the ID. This is because our receiver builds on our prior work [25, 26, 29, 42] and has gone through multiple iterations to improve its reliability.

5.2.2 Evaluating passive Wi-Fi's carrier sense

The plugged-in device performs carrier sense and signals a specific passive Wi-Fi device to transmit. To compare how our mechanism compares to standard Wi-Fi, we compare the performance of a concurrent Wi-Fi transmitter-receiver pair in the presence of a passive Wi-Fi transmitter with that of a traditional Wi-Fi transmitter. We use two Intel 5350 Wi-Fi chipsets to transmit and receive Wi-Fi packets using iperf. The devices use the chipset's default bit rate adaptation. We run experiments in two scenarios: 1) we use a Ralink RT2070 Wi-Fi chipset to transmit packets at 1 Mbps every 15 ms and 2) we set our passive Wi-Fi device to transmit its packet every 15 ms at 1 Mbps using our carrier sense mechanism. We measure the throughput achieved by a concurrent Wi-Fi transmitter-receiver pair in the presence of these two devices. Fig. 17(b) plots the TCP throughput and shows that passive Wi-Fi has a similar impact on the ongoing flow as a traditional Wi-Fi transmitter. This is because, passive Wi-Fi adds only a small fixed 100 μs overhead. This small overhead is however overshadowed by transient changes in network conditions.

5.3 Applications

We first consider low latency sensors like microphones and cameras that transmit continuously. We then analyze duty-cycled sensors.

1) Low power microphones consume 17 μW [1] and an ADC digitizing the microphone output consumes 33 μW [1], resulting in 50 μW for the sensing subsystem. If we use an IoT Wi-Fi chipset by Gainspan or TI to continuously transmit audio, the active Wi-Fi transmitter consumes 670 mW [9, 21]. This results in a total power budget of 670.05 mW which is dominated by the Wi-Fi chipset. However, if we use passive Wi-Fi at 1 Mbps, the power budget drops to 65 μW, i.e., a 1000x reduction.

2) A low power camera like OV7690 operating at VGA resolution and capturing one image per second consumes an average of 10 mW [15]. The camera outputs raw data at 2.45 Mbps which can be transferred wirelessly without power hungry on-board compression. Using an IoT Wi-Fi chipset from Gainspan or TI, brings the total power consumption of the system to 680 mW. If we substitute an active Wi-Fi chipset which consumes 670 mW of power with 11 Mbps passive Wi-Fi, we can improve the battery life of Wi-Fi video camera by at least 50x [9, 21].

3) Duty cycled sensors such as iBeacon [11] and home proximity sensors [16] periodically transmit data using Bluetooth Low Energy and ZigBee protocols respectively. They typically transmit beacons/data packets at a rate of 100 ms to 900 ms and last for 3 months to 3 years respectively on a coin cell battery [11]. If we replace the BLE/ZigBee transmitter which consumes 35 mW [20] in transmit mode with passive Wi-Fi consuming 15 μW, the battery life can be extended well beyond 10 years.

6 Related Work

RFID systems. RFID tags backscatter the signal back to a dedicated 900 MHz RFID reader. The use of backscatter as a general communication mechanism, however, has been limited to RFID systems for two key reasons. First, to decode the weak backscattered signals, the reader eliminates the strong signal from the reader using full-duplex radios [27, 43]. This requires expensive circulators and highly linear analog RF front end at the reader that contributes to its high cost. In contrast, Wi-Fi chipsets do not require the specialized components, can be fully integrated in silicon and hence, are orders of magnitude less expensive. Second, enabling backscatter communication with existing devices requires a complete hardware change to their chipsets and incorporating a dedicated full duplex radio; this is a high bar that has limited the adoption of backscatter beyond RFID.

Wi-Fi and ambient backscatter systems. Since 2013, we have introduced the concepts of ambient and Wi-Fi backscatter [25, 28, 37] where battery-free devices communicate with each other by backscattering ambient signals such as TV and Wi-Fi transmissions. The basic difference between these designs and passive Wi-Fi is that Wi-Fi backscatter systems create an additional narrowband data stream to ride on top of existing Wi-Fi signals. In contrast, passive Wi-Fi aims to use backscatter to generate 802.11b transmissions that can be decoded by billions of existing devices with a Wi-Fi chipsets.

In particular, our prior work on Wi-Fi backscatter [25] demonstrated that existing Wi-Fi chipsets can decode backscattered information from a tag using changes to the per-packet CSI/RSSI values at 1 kbps bitrates and a 2 m range. [38] improved the rate of this communication using a full-duplex radio to cancel the high-power Wi-Fi transmissions from the reader and decode the weak phase-modulated narrowband backscattered signal at the reader. This has allowed them to achieve data rates of up to 5 Mbps at a range of 1 m and 1 Mbps at a range of 5 m. A recent news release [14] claims to achieve 330 Mbps Wi-Fi backscatter communication at 2.5 m using a custom IC that implements a full-duplex radio. The challenge with these full-duplex designs is that they have the same problem as conventional RFID designs— they require a custom full-duplex radio to be incorporated at the receiver and hence the backscattered signals cannot be decoded on any of the existing Wi-Fi devices.

Finally, [23] creates Bluetooth signals using subcarrier modulation to create 370 kHz narrowband 2-FSK signals. Instead, we create 22 MHz DSSS/CCK transmissions using backscatter and enable Wi-Fi transmissions. We also present a network-layer stack design that enables us to operate with existing devices in the ISM band.

Duty-cycled radios. The key idea in these systems is to design a custom low power radio transmitter and use a wakeup receiver to duty cycle the transmitter and reduce the average transceiver power consumption [41]. The power consumption of such transmitters at sub-milliwatt output power is in the order of 100 μW [36, 44] to few mWs [22, 35, 40]. Further, such radios use custom protocols supporting 10-100 kbps data rates that require deployment of special purpose receivers and hardware. In contrast, passive Wi-Fi generates Wi-Fi transmissions at tens of microwatts of power. Given the ubiquity of Wi-Fi, this significantly lowers the bar for adoption. Further, the duty cycle operation is orthogonal to passive Wi-Fi and can be used to further reduce the power consumption of a system employing passive Wi-Fi.

Low power Wi-Fi transceivers. The Wi-Fi industry has designed chipsets for IoT applications including QUALCOMM QCA4002 and QCA4004 [18]. These designs reduce the power consumption by decreasing the transmit power by up to a half when in proximity of another device. They also optimize the power consumption of their sleep mode to be less than 1 mW. Gainspan and TI Wi-Fi chipsets incorporate a 20 μW standby mode and can switch to active mode within tens of milliseconds [9, 21]. However, their active transmission power is around 600 mW [9, 21] which is orders of magnitude higher than passive Wi-Fi. Intel's Moore's radio [7] designs digital versions for RF components such as frequency synthesizers. This reduces the cost and size of the RF chipset rather than its power — a digital Wi-Fi frequency synthesizer consumes 10-50 mW [7, 24] which is similar in power consumption to its analog counterpart.

Finally, recent low power Wi-Fi receiver designs use techniques like dynamic voltage and frequency scaling [30] and compressive sensing [31]. In particular, SloMo [31] leverages the sparsity inherent to 802.11b DSSS signals using compressive sensing to operate the radio at a lower clock rate. Enfold [30] extends this to work with OFDM modulation. Our work on enabling ultra-low power Wi-Fi transmissions is complimentary to this work and can in principle be integrated together.

7 Conclusion

We demonstrate for the first time that one can generate 802.11b transmissions using backscatter communication, while consuming 4-5 orders of magnitude lower power than existing Wi-Fi chipsets. Wi-Fi has traditionally been considered a power-consuming system. Thus, it has not been widely adopting in the sensor network and IoT space where low-power devices primarily transmit data. We believe that, with its orders of magnitude lower power consumption, passive Wi-Fi has the potential to transform the Wi-Fi industry.

Acknowledgements. We thank Ben Ransford and our shepherd Jon Howell. This work was funded in part by NSF under awards CNS-1452494 and CNS-1407583 and a Qualcomm fellowship.

References

[1] ADMP801. http://www.cdiweb.com/datasheets/invensense/ADMP801_2_Page.pdf.

[2] Altera de1 fpga development board. http://www.terasic.com.tw/cgi-bin/page/archive.pl?No=83.

[3] Android wi-fi analyzer. https://play.google.com/store/apps/details?id=com.farproc.wifi.analyzer&hl=en.

[4] Atheros targets cellphone with wi-fi chip. http://www.eetimes.com/document.asp?doc_id=1172134.

[5] Cadence rfspectre. http://www.cadence.com/products/rf/spectre_rf_simulation/pages/default.aspx.

[6] Co-existence of wi-fi and bluetooth radios by marvell. http://www.marvell.com/wireless/assets/Marvell-WiFi-Bluetooth-Coexistence.pdf.

[7] Connecting the future: The latest research from intel labs. http://download.intel.com/newsroom/kits/idf/2012_fall/pdfs/IDF2012_Justin_Rattner.pdf.

[8] Dropcam. https://nest.com/camera/meet-nest-cam/?dropcam=true.

[9] Gainspan gs1500m. http://www.alphamicro.net/media/412417/gs1500m_datasheet_rev_1_4.pdf.

[10] Hms190bms8 by hittite microwave devices. https://www.hittite.com/content/documents/data_sheet/hmc190bms8.pdf.

[11] ibeacons. http://beekn.net/2014/04/will-apple-pull-plug-ibeacon-devices/.

[12] Ieee 802.11 standard, 2012. http://standards.ieee.org/getieee802/download/802.11-2012.pdf.

[13] Max2830 by maxim. https://datasheets.maximintegrated.com/en/ds/MAX2830.pdf.

[14] Nasa news release: A wi-fi reflector chip to speed up wearables. http://www.jpl.nasa.gov/news/news.php?feature=4663.

[15] Ovm 7690 camera module. http://www.ovt.com/uploads/parts/OVM7690_PB(1.0)_web.pdf.

[16] Proximity sensors. https://www.ia.omron.com/products/category/sensors/proximity-sensors/.

[17] Qualcomm atheros 9462. http://www.qca.qualcomm.com/wp-content/uploads/2013/11/AR9462.pdf.

[18] Qualcomm qca4002 and qca4004. http://www.eeworld.com.cn/zt/wireless/downloads/QCA4002-4004FIN.pdf.

[19] Synopsis design complier. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx.

[20] TI CC2541. http://www.ti.com/lit/ds/symlink/cc2541.pdf.

[21] TI CC3100MOD. http://www.ti.com/lit/ds/symlink/cc3100mod.pdf.

[22] J. Ayers, N. Panitantum, K. Mayaram, and T. S. Fiez. A 2.4 ghz wireless transceiver with 0.95 nj/b link energy for multi-hop battery-freewireless sensor networks. In *VLSI Circuits (VLSIC), 2010 IEEE Symposium on*, pages 29–30. IEEE, 2010.

[23] J. Ensworth and M. Reynolds. Every smart phone is a backscatter reader: Modulated backscatter compatibility with bluetooth 4.0 low energy (ble) devices. In *RFID, 2015 IEEE International Conference on*.

[24] K. Greene. Intel's tiny wi-fi chip could have a big impact. *MIT Technology review*, 2012.

[25] B. Kellogg, A. Parks, S. Gollakota, J. R. Smith, and D. Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.

[26] B. Kellogg, V. Talla, and S. Gollakota. Bringing gesture recognition to all devices. In *Usenix NSDI*, volume 14, 2014.

[27] P. B. Khannur, X. Chen, D. L. Yan, D. Shen, B. Zhao, M. K. Raja, Y. Wu, R. Sindunata, W. G. Yeoh, and R. Singh. A universal uhf rfid reader ic in 0.18-μm cmos technology. *Solid-State Circuits, IEEE Journal of*, 43(5):1146–1155, 2008.

[28] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[29] V. Liu, V. Talla, and S. Gollakota. Enabling instantaneous feedback with full-duplex backscatter. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 67–78. ACM, 2014.

[30] F. Lu, P. Ling, G. M. Voelker, and A. C. Snoeren. Enfold: downclocking ofdm in wifi. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 129–140. ACM, 2014.

[31] F. Lu, G. M. Voelker, and A. C. Snoeren. Slomo: Downclocking wifi communication. In *NSDI*, pages 255–258, 2013.

[32] J. Manweiler and R. Roy Choudhury. Avoiding the rush hours: Wifi energy management via traffic isolation. In *MobiSys*, 2011.

[33] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *MobiCom*, 2012.

[34] P. Nikitin and K. Rao. Theory and measurement of backscattering from RFID tags. *Antennas and Propagation Magazine, IEEE*, 48(6):212 –218, december 2006.

[35] B. Otis, Y. Chee, R. Lu, N. Pletcher, and J. Rabaey. An ultra-low power mems-based two-channel transceiver for wireless sensor networks. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 20–23. IEEE, 2004.

[36] J. Pandey and B. P. Otis. A sub-100 w mics/ism band transmitter based on injection-locking and frequency multiplication. *Solid-State Circuits, IEEE Journal of*, 46(5):1049–1058, 2011.

[37] A. N. Parks, A. Liu, S. Gollakota, and J. R. Smith. Turbocharging ambient backscatter communication. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.

[38] D. Pharadia, K. R. Joshi, M. Kotaru, and S. Katti. Backfi: High throughput wifi backscatter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[39] J. G. Proakis and M. Salehi. Digital communications. 2005. *McGraw-Hill, New York.*

[40] J. Rabaey, J. Ammer, B. Otis, F. Burghardt, Y. Chee, N. Pletcher, M. Sheets, and H. Qin. Ultra-low-power design. *Circuits and Devices Magazine, IEEE*, 22(4):23–29, 2006.

[41] J. M. Rabaey, M. J. Ammer, J. L. da Silva, D. Patel, and S. Roundy. Picoradio supports ad hoc ultra-low power wireless networking. *Computer*, 33(7):42–48, 2000.

[42] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. Design of an rfid-based battery-free programmable sensing platform. *Instrumentation and Measurement, IEEE Transactions on*, 57(11):2608–2615, 2008.

[43] C. Ying and Z. Fu-Hong. A system design for uhf rfid reader. In *Communication Technology, 2008. ICCT 2008. 11th IEEE International Conference on*, pages 301–304. IEEE, 2008.

[44] F. Zhang, Y. Zhang, J. Silver, Y. Shakhsheer, M. Nagaraju, A. Klinefelter, J. Pandey, J. Boley, E. Carlson, A. Shrivastava, et al. A batteryless 19μw mics/ism-band energy harvesting body area sensor node soc. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 298–300. IEEE, 2012.

Decimeter-Level Localization with a Single WiFi Access Point

Deepak Vasisht[†], Swarun Kumar[‡], Dina Katabi[†]
[†]MIT CSAIL, [‡] CMU
deepakv@mit.edu, swarun@cmu.edu, dk@mit.edu

Abstract – We present Chronos, a system that enables a single WiFi access point to localize clients to within tens of centimeters. Such a system can bring indoor positioning to homes and small businesses which typically have a single access point.

The key enabler underlying Chronos is a novel algorithm that can compute sub-nanosecond time-of-flight using commodity WiFi cards. By multiplying the time-of-flight with the speed of light, a MIMO access point computes the distance between each of its antennas and the client, hence localizing it. Our implementation on commodity WiFi cards demonstrates that Chronos's accuracy is comparable to state-of-the-art localization systems, which use four or five access points.

1. Introduction

Recent years have seen significant advances in indoor positioning using wireless signals [48, 28]. State-of-the-art systems have achieved an accuracy of tens of centimeters, even using commodity WiFi chipsets [30, 32, 18]. Existing proposals however target enterprise networks, where multiple WiFi access points can combine their information and cooperate together to locate a user. However, the vast majority of homes and small businesses today have a single WiFi access point. Consequently, this large constituency of wireless networks has been left out of the benefits of accurate indoor positioning.

Developing a technology that can locate users and objects using a single WiFi access point would enable a range of important applications:

(i) *Smart Home Occupancy:* In particular, indoor positioning can play a crucial role in the smart home vision, where WiFi enabled home automation systems like NEST are gaining increasing popularity [37]. Accurate localization addresses a long-standing problem in home automation: reliable occupancy detection [36, 6]. With WiFi-based localization, one can track the number of users per room using their phones or wearables, and accordingly adapt heating and lighting. Knowing the identity of these occupants can then help personalize heating and lighting levels based on user preferences.

(ii) *WiFi Geo-fencing:* Beyond the home, indoor positioning can benefit small businesses that use a single access point to offer free WiFi to attract customers. But with increasingly congested networks, business owners seek to restrict WiFi connectivity to their own customers, given that 32% of users in the US admit to have accessed open WiFi networks outside the premises they serve [47]. Yet securing these networks with passwords is inconvenient, both to customers that connect to these networks and the business owners who must frequently change the passwords. Indoor positioning with a single access point provides a natural solution to this problem because it can automatically authenticate customers based on their location.

(iii) *Device-to-device Location:* More generally, enabling two WiFi nodes to localize each other without additional infrastructure support has implications in areas where WiFi networks may not exist altogether. Imagine traveling with friends or family in countries where WiFi is not as prevalent as in the US, yet still be able to find each other in a mall, museum, or train station, without the need to connect to a WiFi infrastructure.

Our goal is to design a system that enables a single WiFi node (e.g., an access point) to localize another, without support from additional infrastructure. Further, we would like a design that works on commodity WiFi NICs and does not require any additional sensors (cameras, accelerometers, etc.).

As we design for the above goal, it helps to first examine why past systems need multiple access points. The most direct approach to RF-based positioning estimates the time-of-flight (i.e., propagation time) and multiplies it by the speed of light to obtain the distance [23, 16]. However, past proposals for WiFi-based positioning cannot measure the *absolute* time-of-flight. They measure only *differences* in the time-of-flight across the receiver's antennas. Such time differences allow those systems to infer the direction of the source with respect to the receiver, known as the angle of arrival (AoA) [48]. But they don't provide the distance between the source and the receiver. Thus, past work has to intersect the direction of the source from multiple access points to localize it. In fact, past proposals typically use four or five access points to achieve tens of centimeters accuracy [30, 32, 48, 50]. Even the few recent proposals to localize using one WiFi access point [35, 53] require users to walk to multiple locations to emulate the presence of multiple access points. They then intersect signal measurements across these locations coupled with accelerometer readings to infer the user's trajectory.

There are however non-WiFi systems that can accurately measure the absolute time-of-flight, and hence localize using a single receiver. Such systems use specialized ultra wideband radios that span multiple GHz [5, 41]. Since time resolution is inversely related to the radio bandwidth, such devices can measure time-of-flight at sub-nanosecond accuracy, and hence localize an object to within tens of centimeters. In contrast, directly measuring time with a 20MHz or 40MHz WiFi radio results in errors of 7 to 15 meters [30].

Motivated by the above analysis, we investigated whether a WiFi radio can emulate a wideband multi-GHz radio, for the purpose of localization. Our investigation led to Chronos, an indoor positioning system that enables a pair of WiFi devices to localize each other. It runs on commodity WiFi cards, and does not require any external sensor (e.g., accelerometer, or camera). Chronos works by making a WiFi card emulate a very wideband radio. In particular, while each WiFi frequency band is only tens of Megahertz wide, there are many such bands that together span a very wide bandwidth. Chronos therefore transmits packets on multiple WiFi bands and stitches their information together to give the illusion of a wideband radio.

Yet, emulating a wideband radio using packets transmitted on different frequency bands is not easy. Stitching measurements across such packets requires Chronos to overcome three challenges:

Resolving Phase Offsets: First, to emulate a wideband radio, Chronos needs to stitch channel state information (CSI) captured by multiple packets, transmitted in different WiFi frequency bands, at different points in time. However, the very act of hopping between WiFi frequency bands introduces a random initial phase offset as the hardware resets to each new frequency (i.e., PLL locking). Chronos must therefore recover time-of-flight to perform positioning despite these random phase offsets.

Eliminating Packet Detection Delay: Second, any measurement of time-of-flight of a packet necessarily includes the delay in detecting its presence. Different packets however experience different random detection delays. To make matters worse, this packet detection delay is typically orders-of-magnitude higher than time-of-flight. For indoor WiFi environments, time-of-flight is just a few nanoseconds, while packet detection delay spans hundreds of nanoseconds [38]. Chronos must tease apart the time-of-flight from this detection delay.

Combating Multipath: Finally, in indoor environments, signals do not experience a single time-of-flight, but a time-of-flight spread. This is because RF signals in indoor environments bounce off walls and furniture, and reach the receiver along multiple paths. As a result, the receiver obtains several copies of the signal, each having experienced a different time-of-flight. To perform accurate localization, Chronos therefore must disentangle the time-of-flight of the direct path from all the remaining paths.

The body of this paper explains how Chronos overcomes these challenges, computes the absolute time-of-flight, and enables localization using a single access point.

Summary of Results: We have implemented Chronos and evaluated its performance on devices equipped with Intel 5300 WiFi cards. Our results reveal the following:

- Chronos computes the time-of-flight with a median error of 0.47 ns in line-of-sight and 0.69 ns in non-line-of-sight settings. This corresponds to a median distance error of 14.1 cm and 20.7 cm respectively.
- Chronos enables a WiFi device (e.g., an AP) to localize another with a median error of 65 cm in line-of-sight and 98 cm in non-line-of-sight settings.

To demonstrate Chronos's capabilities, we use it for three applications:

- *Smart Home Occupancy:* Chronos can be used to track the number of occupants in different rooms of a home using a single access point – a key primitive for smart homes that adapt heating and lighting. Experiments conducted in a 2-bedroom apartment with 4 occupants show that Chronos maps residents in a home to the correct room they are in with an accuracy of 94.3%.
- *WiFi Geo-fencing:* Chronos can be used by small businesses with a single access point to restrict WiFi connectivity to customers within their facility. Experiments in a coffee house reveal that Chronos achieves this to an accuracy of 97%.
- *Personal Drone:* Chronos's ability to locate a pair of user devices can directly benefit the navigation systems of personal robots such as recreational drones. Chronos enables personal drones that can maintain a safe distance from their user by tracking their owner's handheld device. Our experiments using an AscTec Quadrotor reveal that it maintains the required distance relative to a user's device with a root mean-squared error of 4.2 cm.

Contributions: To our knowledge, Chronos is the first system that enables a node with a commercial WiFi card to locate another at tens of centimeters accuracy without any third party support, be it other WiFi nodes or external sensors (e.g., accelerometers). Chronos also contributes the first algorithm for measuring the absolute time-of-flight on commercial WiFi cards at sub-nanosecond accuracy.

2. Overview

We briefly outline the organization of the rest of this paper. Chronos localizes a pair of WiFi devices without third party support by computing time of flight of signals between them. Sec. §3 describes our approach to compute time-of-flight by stitching together information across multiple WiFi frequency bands. It is followed by a description of the challenges faced by Chronos and how it addresses them. Specifically:

- **Eliminating Packet Detection Delay:** First, Chronos disentangles the time-of-flight from packet detection

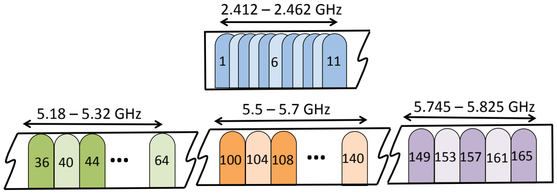


Figure 1: **WiFi Bands:** Depicts WiFi bands at 2.4 GHz and 5 GHz. Note that some of these frequencies (e.g. 5.5-5.7 GHz) are DFS bands in the U.S. that many 802.11h compatible 802.11n radios like Intel 5300 support.

delay, since the latter has no connection to the distance between transmitter and receiver (See Sec. §4).

- **Combating Multipath:** Second, Chronos separates the time-of-flight of the direct path of the wireless signal from that of all the remaining paths (See Sec. §5).
- **Resolving Phase Offsets:** Finally, Chronos removes arbitrary phase offsets that are introduced as the WiFi receiver hops between frequency bands (See Sec. §6).

3. Measuring Time of Flight

In this section, we describe how Chronos measures accurate time-of-flight of signals between a pair of WiFi devices without third party support. For clarity, the rest of this section assumes signals propagate from the transmitter to a receiver along a single path with no detection delay or phase offsets. We address challenges stemming from packet detection delay, multipath and phase offsets in §4, §5 and §6 respectively.

Chronos's approach is based on the following observation: Conceptually, if our receiver had a very wide bandwidth, it could readily measure time-of-flight from a single receiving device at a fine-grained resolution (since time and bandwidth are inversely related). Unfortunately, today's WiFi devices do not have such wide bandwidth. But there is another opportunity: WiFi devices are known to span multiple frequency bands scattered around 2.4 GHz and 5 GHz. Combined, these bands span almost one GHz of bandwidth. By making a transmitter and receiver hop between these different frequency bands, we can gather many different measurements of the wireless channel. We can then "stitch together" these measurements to compute the time-of-flight, as if we had a very wideband radio.

However, our method for stitching time measurements across WiFi frequency bands must account for the fact that many WiFi bands are non-contiguous, unequally spaced, and even multiple GHz apart (Fig. 1). Chronos overcomes these issues by exploiting the relation between the time-of-flight and the phase of wireless channels. Specifically, we know from basic electromagnetics that as a signal propagates in time, it accumulates a corresponding phase depending on its frequency. The higher the frequency of the signal, the faster the phase accumulates. To illustrate, let us consider a transmitter sending a signal to its receiver. Then we can write the wireless channel h as [42]:

$$h = ae^{-j2\pi f\tau}, \tag{1}$$

where a is the signal magnitude, f is the frequency and τ is the time-of-flight. The phase of this channel depends on time-of-flight as:

$$\angle h = -2\pi f\tau \mod 2\pi \tag{2}$$

Notice that the above equation depends directly on the signal's time-of-flight and hence, we can use it to measure the time-of-flight τ as:

$$\tau = -\frac{\angle h}{2\pi f} \mod \frac{1}{f} \tag{3}$$

The above equation gives us the time-of-flight modulo $1/f$. Hence, for a WiFi frequency of 2.4 GHz, we can only obtain the time-of-flight *modulo* 0.4 nanoseconds. Said differently, transmitters with times-of-flight 0.1 ns, 0.5 ns, 0.9 ns, 1.3 ns, etc. all produce identical phase in the wireless channel. In terms of physical distances, this means transmitters at distances separated by multiples of 12 cm (e.g., 3 cm, 15 cm, 27 cm, 39 cm, etc.) all result in the same channel phase. Consequently, there is no way to distinguish between these transmitters using their phase on a single frequency band.

Indeed, this is precisely why Chronos needs to hop between multiple frequency bands $\{f_1, \ldots, f_n\}$ and measure the corresponding wireless channels $\{h_1, \ldots, h_n\}$. The result is a system of equations, one per frequency, that measure the time-of-flight modulo different values:

$$\forall i \in \{1, 2, \ldots, n\} \quad \tau = -\frac{\angle h_i}{2\pi f_i} \mod \frac{1}{f_i} \tag{4}$$

Notice that the above set of equations has the form of the well-known Chinese remainder theorem [45]. Such equations can be readily solved using standard modular arithmetic algorithms, even amidst noise [14] and have been used in prior work, in the context of range estimation ([44, 43]).[1] The theorem states that solutions to these equations are unique modulo a much larger quantity – the Least Common Multiple (LCM) of $\{1/f_1, \ldots, 1/f_n\}$.

To illustrate how the above system of equations works, consider a source at 0.6 m whose time-of-flight is 2 ns. Say the receiver measures the channel phases from this source on five candidate WiFi frequency bands as shown in Fig. 2. We note that a measurement on each of these channels produces a unique equation for τ, like in Eqn. 4. Each equation has multiple solutions, depicted as colored vertical lines in Fig. 2. However, only the correct solution of τ will satisfy all equations. Hence, by picking the solution satisfying the most number of equations (i.e., the τ with most number of aligned lines in Fig. 2), we can recover the true time-of-flight of 2 ns.

Note that our solution based on the Chinese remainder theorem makes no assumptions on whether the set

[1] Algorithm 1 in §5 provides a more general version of Chronos's algorithm to do this while accounting for noise and multipath

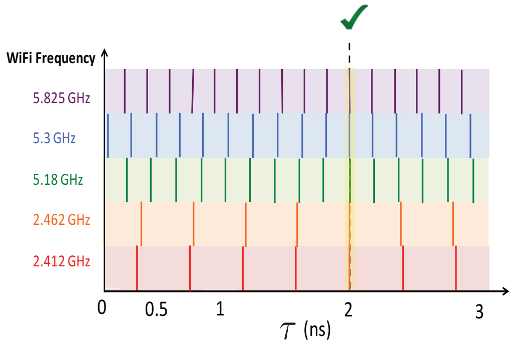


Figure 2: **Measuring Time-of-Flight:** Consider a wireless transmitter at a distance of 0.6 m, i.e. a time-of-flight of 2 ns. The phase of each WiFi channel results in multiple solutions, depicted as colored lines, including 2 ns. However, the solution that satisfies most equations, i.e. has the most number of aligned colored lines is the true time-of-flight (2 ns).

of frequencies $\{f_1, \ldots, f_n\}$ are equally separated or otherwise. In fact, having unequally separated frequencies makes them less likely to share common factors, boosting the LCM. Thus, counter-intuitively, the scattered and unequally-separated bands of WiFi (Fig. 1) are not a challenge, but an opportunity to resolve larger values of τ.

While the above provides a mathematical formulation of our algorithm, we describe below important systems considerations when dealing with commercial WiFi cards:

- Chronos must ensure both the WiFi transmitter and receiver hop synchronously between multiple WiFi frequency bands. Chronos achieves this using a frequency band hopping protocol driven by the transmitter. Before switching frequency bands (every 2-3 ms in our implementation), the transmitter issues a control packet that advertises the frequency of the next band to hop to. The receiver responds with an acknowledgment and switches to the advertised frequency. Once the acknowledgment is received, the transmitter switches frequency bands as well. As a fail-safe, transmitters and receivers revert to a default frequency band if they do not receive packets or acknowledgments from each other for a given time-out duration on any band.
- Our implementation of Chronos sweeps all WiFi bands in 84 ms (12 times per second). This is within the channel coherence time of indoor environments [39] and can empirically localize users at walking speeds (§10.3).
- Finally, we discuss and evaluate the implications of Chronos's protocol on data traffic in §9.3.

4. Eliminating Packet Detection Delay

So far, we computed time-of-flight based on the channels h_i, that signals experience when transmitted over the air on different frequencies f_i. In practice however, there is a difference between the channel over the air, h_i, and the channel as measured by the receiver, $\tilde{h}_i$. Specifically, the *measured* channel at the receiver, $\tilde{h}_i$, experiences a delay in addition to time-of-flight: the delay in detecting the presence of a packet. This delay occurs because WiFi receivers detect the presence of a packet based on the energy of its first few time samples. The number of samples that the receiver needs to cross its energy detection threshold varies based on the power of the received signal, as well as noise. While this variation may seem small, packet detection delays are often an *order-of-magnitude* larger than time-of-flight, particularly in indoor environments, where time-of-flight is just a few tens of nanoseconds (See §9.1). Hence, accounting for packet detection delay is crucial for accurate time-of-flight and distance measurements.

Thus, our goal is to derive the true channel h_i (which incorporates the time-of-flight alone) from the measured channel $\tilde{h}_i$ (which incorporates both time-of-flight and packet detection delay). To do this, we exploit the fact that WiFi uses OFDM. Specifically, the bits of WiFi packets are transmitted in the frequency domain on several small frequency bins called OFDM subcarriers. This means that the wireless channels $\tilde{h}_i$ can be measured on each subcarrier. We then make the following claim:

Claim 4.1. *The measured channel at subcarrier-*0 *does not experience packet detection delay, i.e., it is identical in phase to the true channel at subcarrier* 0.

To see why this claim holds, note that while time-of-flight and packet detection delay appear very similar, they occur at different stages of a signal's lifetime. Specifically, time-of-flight occurs while the signal is transmitted over the air (i.e., in passband). In contrast, packet detection delay stems from energy detection that occurs in digital processing once the carrier frequency has been removed (in baseband). Thus, time-of-flight and packet detection delay affect the wireless OFDM channels in different ways.

To understand this difference, consider the WiFi frequency band, i. Let $\tilde{h}_{i,k}$ be the measured channel of OFDM subcarrier k, at frequency $f_{i,k}$. $\tilde{h}_{i,k}$ experiences two phase rotations in different stages of the signal's lifetime:

- A phase rotation in the air proportional to the over-the-air frequency $f_{i,k}$. From Eqn. 2 in §3, this phase value for a frequency $f_{i,k}$ is:
$$\angle h_{i,k} = -2\pi f_{i,k}\tau \mod 2\pi,$$
where τ is the time-of-flight.
- An additional phase rotation due to packet detection after the removal of the carrier frequency. This additional phase rotation can be expressed as:
$$\Delta_{i,k} = -2\pi(f_{i,k} - f_{i,0})\delta_i,$$
where δ_i is the packet detection delay.

Thus, the total measured channel phase at subcarrier k is:

$$\angle\tilde{h}_{i,k} = (\angle h_{i,k} + \Delta_{i,k}) \mod 2\pi \quad (5)$$

$$= (-2\pi f_{i,k}\tau - 2\pi(f_{i,k} - f_{i,0})\delta_i) \mod 2\pi \quad (6)$$

Notice from the above equation that the second term $\Delta_{i,k} = -2\pi(f_{i,k} - f_{i,0})\delta_i = 0$ at $k = 0$. In other words, at the zero-subcarrier of OFDM, the measured channel $\tilde{h}_{i,k}$ is identical in phase to the true channel $h_{i,k}$ over-the-air which validates our claim.

In practice, this means that we can apply the Chinese Remainder theorem as described in Eqn. 4 of §3 at the zero-subcarriers (i.e. center frequencies) of each WiFi frequency band. In the U.S., WiFi at 2.4 GHz and 5 GHz has a total of 35 WiFi bands with independent center frequencies.[2] Therefore, a sweep of all WiFi frequency bands results in 35 independent equations like in Eqn. 4, which we can solve to recover time-of-flight.

One problem still needs to be addressed. So far we have used the measured channel at the zero-subcarrier of WiFi bands. However, WiFi transmitters do not send data on the zero-subcarrier, meaning that this channel simply cannot be measured. This is because the zero-subcarrier overlaps with DC offsets in hardware that are extremely difficult to remove [22, 3]. So how can one measure channels on zero-subcarriers if they do not even contain data?

Fortunately, Chronos can tackle this challenge by using the remaining WiFi OFDM subcarriers, where signals are transmitted. Specifically, it leverages the fact that indoor wireless channels are based on physical phenomena. Hence, they are continuous over a small number of OFDM subcarriers [27]. This means that Chronos can interpolate the measured channel phase across all subcarriers to estimate the missing phase at the zero-subcarrier.[3] Indeed, the 802.11n standard [3] measures wireless channels on as many as 30 subcarriers in each WiFi band. Hence, interpolating between the channels not only helps Chronos retrieve the measured channel on the zero-subcarrier, but also provides additional resilience to noise.

To summarize, Chronos applies the following steps to account for packet detection delay: (1) It obtains the measured wireless channels on the 30 subcarriers on the 35 available WiFi bands; (2) It interpolates between these subcarriers to obtain the measured channel phase on the zero-subcarriers on each of these bands, which is unaffected by packet detection delay. (3) It retrieves the time-of-flight using the resulting 35 channels.

5. Combating Multipath

So far, our discussion has assumed that a wireless signal propagates along a single direct path between its transmitter and receiver. However, indoor environments are rich in multipath, causing wireless signals to bounce off objects in the environment like walls and furniture. Fig. 3(a) illustrates an example where the signal travels along three paths from its sender to receiver. The signals on each of these paths propagate over the air incurring different time delays as well as different attenuations. The ultimate received signal is therefore the sum of these multiple signal copies, each having experienced a different propagation delay. Fig. 3(b) represents this using a *multipath profile*. This profile has peaks at the propagation delays of signal paths, scaled by their respective attenuations. Hence, Chronos needs a mechanism to find such a multipath profile, so as to separate the propagation delays of different signal paths. This allows it to then identify the time-of-flight as the least of these propagation delays, i.e. the delay of the most direct (shortest) path.

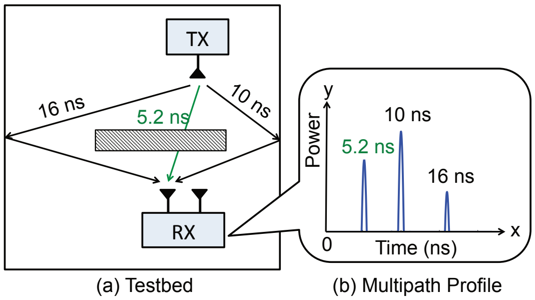


Figure 3: **Combating Multipath:** Consider a signal propagating from a transmitter to a receiver along 3 paths as shown in (a): an attenuated direct path and two reflected paths of lengths 5.2 ns, 10 ns and 16 ns respectively. These paths can be separated by using the inverse discrete Fourier Transform as shown in (b). The plot has 3 peaks corresponding to the propagation delays of the paths, with peak magnitudes scaled by relative attenuations.

5.1 Computing Multipath Profiles

Say that wireless signals from a transmitter reach a receiver along p different paths. The received signal from each path corresponds to amplitudes $\{a_1, \ldots, a_p\}$ and propagation delays $\{\tau_1, \ldots, \tau_p\}$. Observe that Eqn. 1 considers only a single path experiencing propagation delay and attenuation. In the presence of multipath, we can extend this equation to write the measured channel $\tilde{h}_{i,0}$ on center-frequency $f_{i,0}$ as the sum of the channels on each of these paths, i.e.:

$$\tilde{h}_{i,0} = \sum_{k=1}^{p} a_k e^{-j2\pi f_{i,0}\tau_k} \quad , \text{for } i = 1, \ldots, n \tag{7}$$

Now, we need to disentangle these different paths and recover their propagation delays. To do this, notice that the above equation has a familiar form – it is the well-known Discrete Fourier Transform. Thus, if one could obtain the channel measurements at many uniformly-spaced frequencies, a simple inverse-Fourier transform would separate individual paths. Such an inverse Fourier transform has a closed-form expression that can be used to obtain the propagation delay of all paths and compute the multipath profile (up to a resolution defined by the bandwidth).

WiFi frequency bands, however, are not equally spaced – they are scattered around 2.4 GHz and multiple non-contiguous chunks at 5 GHz, as shown in Fig. 1. While we

[2] Including the DFS bands at 5 GHz in the U.S. which are supported by many 802.11h-compatible 802.11n radios, e.g., the Intel 5300.

[3] Our implementation of Chronos uses cubic spline interpolation.

can measure $\tilde{h}_{i,0}$ at each WiFi band, these measurements will not be at equally spaced frequencies and hence cannot be simply used to compute the inverse Fourier transform. In fact, since our measurements of the channels are not uniformly spaced, we are dealing with the *Non-uniform* Discrete Fourier Transform or NDFT [8]. To recover the multipath profile, we need to invert the NDFT.

5.2 Inverting the NDFT

The NDFT is an under-determined system, where the responses of multiple frequency elements are unavailable [19, 15]. Thus, the inverse of such a Fourier transform does not have a single closed-form solution, but several possible solutions. So how can Chronos pick the best among those solutions to find the true times-of-flight?

Chronos adds another constraint to the inverse-NDFT optimization. Specifically, this constraint favors solutions that are sparse, i.e., have few dominant paths. Intuitively, this stems from the fact that while signals in indoor environments traverse several paths, a few paths tend to dominate as they suffer minimal attenuation [10].[4] Indeed other localization systems make this assumption as well, albeit less explicitly. For instance, antenna-array systems can resolve a limited number of dominant paths based on the number of antennas they use.

We can formulate the sparsity constraint mathematically as follows. Let the vector **p** sample inverse-NDFT at m discrete values $\tau \in \{\tau_1, \ldots, \tau_m\}$. Then, we can introduce sparsity as a simple constraint in the NDFT inversion problem that minimizes the L-1 norm of **p**. Indeed, it has been well-studied in optimization theory that minimizing the L-1 norm of a vector favors sparse solutions for that vector [7]. Thus, we can write the optimization problem to solve for the inverse-NDFT as:

$$\min \|\mathbf{p}\|_1 \tag{8}$$

$$\text{s.t.} \quad \|\tilde{\mathbf{h}} - \mathcal{F}\mathbf{p}\|_2^2 = 0 \tag{9}$$

where, $\mathcal{F}$ is the $n \times m$ Fourier matrix, i.e. $\mathcal{F}_{i,k} = e^{-j2\pi f_{i,0}\tau_k}$, $\tilde{\mathbf{h}} = [\tilde{h}_{1,0}, \ldots, \tilde{h}_{n,0}]^T$ is the $n \times 1$ vector of wireless channels at the n different center-frequencies $\{f_{1,0}, \ldots, f_{n,0}\}$, $\|\cdot\|_1$ is the L-1 norm, and $\|\cdot\|_2$ is the L-2 norm. Here, the constraint makes sure that the Discrete Fourier Transform of **p** is $\tilde{\mathbf{h}}$, as desired. In other words, it ensures **p** is a candidate inverse-NDFT solution of $\tilde{\mathbf{h}}$. The objective function favors sparse solutions by minimizing the L-1 norm of **p**.

We can re-formulate the above optimization problem using the method of Lagrange multipliers as:

$$\min_{\mathbf{p}} \|\tilde{\mathbf{h}} - \mathcal{F}\mathbf{p}\|_2^2 + \alpha\|\mathbf{p}\|_1 \tag{10}$$

Notice that the factor α is a sparsity parameter that enforces the level of sparsity. A bigger choice of α leads to fewer non-zero values in **p**.

This objective function is convex but not differentiable.

1 Algorithm to Compute Inverse NDFT

```
▷ Given: Measured Channels, $\tilde{\mathbf{h}}$
▷ $\mathcal{F}$: Non-uniform DFT matrix, such that $\mathcal{F}_{i,k} = e^{-j2\pi f_{i,0}\tau_k}$
▷ $\alpha$: Sparsity parameter; $\epsilon$: Convergence Parameter
▷ Output: Inverse-NDFT, $\mathbf{p}$
▷ Initialize $\mathbf{p}_0$ to a random value, $t = 0, \gamma = \frac{1}{||\mathcal{F}||_2}$.
while $converged = false$ do
    $\mathbf{p}_{t+1}$ =SPARSIFY($\mathbf{p}_t - \gamma\mathcal{F}^*(\mathcal{F}\mathbf{p}_t - \tilde{\mathbf{h}}), \gamma\alpha$)
    if $||\mathbf{p}_{t+1} - \mathbf{p}_t||_2 < \epsilon$ then
        $converged = true$
        $\mathbf{p} = \mathbf{p}_{t+1}$
    else
        $t = t + 1$
    end if
end while
function SPARSIFY($\mathbf{p}$,$t$)
    for $i = 1, 2, ...length(\mathbf{p})$ do
        if $|\mathbf{p}_i| < t$ then
            $\mathbf{p}_i = 0$
        else
            $\mathbf{p}_i = \mathbf{p}_i \frac{|\mathbf{p}_i| - t}{|\mathbf{p}_i|}$
        end if
    end for
end function
```

Our approach to optimize for it borrows from proximal gradient methods, a special class of optimization algorithms that have provable convergence guarantees [24]. Specifically, our algorithm takes as inputs the measured wireless channels $\tilde{\mathbf{h}}$ at the frequencies $\{f_{1,0}, \ldots, f_{n,0}\}$ and the sparsity parameter α. It then applies a gradient-descent style algorithm by computing the gradient of differentiable terms in the objective function (i.e., the L-2 norm), picking sparse solutions along the way (i.e., enforcing the L-1 norm). Algorithm 1 summarizes the steps to invert the NDFT and find the multipath profile.[5]

Inverting the NDFT provides Chronos with the time-of-flight on all paths. Chronos still needs to identify the direct path to compute the distance between transmitter and receiver. To do this, Chronos leverages that: of all the paths of the wireless signal, the direct path is the shortest. Hence, the time-of-flight of the direct path is the time corresponding to the first peak in the multipath profile.

It is worth noting that by making the sparsity assumption, we lose the propagation delays of extremely weak paths in the multipath profile. However, Chronos only needs the propagation delay of the direct path. As long as this path is among the dominant signal paths, Chronos can retrieve it accurately. Of course, in some unlikely scenarios, the direct path may be too attenuated, which leads to poorer localization in that instance. Our results in §9.1 depict the sparsity of representative multipath profiles, and show its impact on overall accuracy.

6. Correcting for Phase Offsets

To work with practical WiFi radios, Chronos has to ad-

[4] We empirically evaluate the sparsity of indoor multipath profiles in typical line-of-sight and non-line-of-sight settings in §9.1.

[5] MATLAB implementation of this algorithm takes 3.1 *s* (standard deviation 0.6 *s*) for Chronos's implementation in Sec. 8.

dress their inherent phase and frequency offsets:

- **PLL Phase Offset:** Frequency hopping causes a random phase offset in the measured channel. This is because the phase-locked loop (PLL) responsible for generating the center frequency for the transmitter and the receiver starts at random initial phase (say, $\phi_{i,0}^{tx}$ and $\phi_{i,0}^{rx}$ respectively). As a result, the channel measured at the receiver is corrupted by an additional phase offset $\phi_{i,0}^{tx} - \phi_{i,0}^{rx}$. This phase offset, if left uncorrected, could render the phase information uncorrelated with the time-of-flight of the signal.
- **Carrier Frequency Offset:** This offset occurs due to small differences in the carrier frequency of the transmitting and receiving radio. This leads to a time varying phase offset across each frequency band. Such differences accumulate quickly over time and need to be corrected for every WiFi packet. Mathematically, in the i^{th} WiFi frequency band, the receiver center frequency $f_{i,0}^{rx}$ is slightly different from the transmitter center frequency, $f_{i,0}^{tx}$. As a result, the channel measurements at the receiver have an additional phase change which is proportional to $f_{i,0}^{tx} - f_{i,0}^{rx}$.

Let us refer to the channel values that incorporate phase and frequency offsets as CSI (channel state information), which is the typical term use in communication systems. Then, the CSI measured at the receiver for the i^{th} frequency band can be written as:

$$\mathrm{CSI}_{i,0}^{rx}(t) = \tilde{h}_{i,0} e^{j(f_{i,0}^{tx} - f_{i,0}^{rx})t + j(\phi_{i,0}^{tx} - \phi_{i,0}^{rx})} \tag{11}$$

So how do we remove the phase and frequency offsets from CSI? To address this issue, Chronos exploits that, the phase and frequency offsets measured on one node with respect to another change sign when measured on the second node with respect to the first. Thus, if one would measure the CSI on the transmitter with respect to the receiver, it would take the following value:

$$\mathrm{CSI}_{i,0}^{tx}(t) = \tilde{h}_{i,0} e^{j(f_{i,0}^{rx} - f_{i,0}^{tx})t + j(\phi_{i,0}^{rx} - \phi_{i,0}^{tx})}. \tag{12}$$

Note that the channel, $\tilde{h}_{i,0}$, in equations 11 and 12 is the same due to reciprocity [20]. We can therefore multiply the CSI measurements at the receiver and the transmitter to recover the wireless channel as follows:

$$\tilde{h}_{i,0}^2 = \mathrm{CSI}_{i,0}^{rx}(t)\mathrm{CSI}_{i,0}^{tx}(t) \tag{13}$$

One may wonder how Chronos measure the CSI at the transmitter. Note however that as part of our channel hopping protocol both nodes have to transmit packets to each other. Hence, the CSI can be measured on both sides and exchanged to apply Eqn. 13.

The above formulation helps us only retrieve the square of the wireless channels $\tilde{h}_{i,0}^2$. However, this is not an issue: Chronos can directly feed $\tilde{h}_{i,0}^2$ into its algorithm (Alg. 1 in §5) instead of $\tilde{h}_{i,0}$. Then the first peak of the resulting multipath profile will simply be at twice the time-of-flight.

To see why, let us look at a simple example. Consider a transmitter and receiver obtaining their signals along two paths, with propagation delays 2 ns and 4 ns. We can write the square of the resulting wireless channels from Eqn. 7 for frequency band i in a simple form:

$$\begin{aligned} \tilde{h}_{i,0}^2 &= (a_1 e^{-j2\pi f_{i,0} \times 2} + a_2 e^{-j2\pi f_{i,0} \times 4})^2 \\ &= a_1^2 e^{-j2\pi f_{i,0} \times 2 \times 2} + 2a_1 a_2 e^{-j2\pi f_{i,0} \times (2+4)} + a_2^2 e^{-j2\pi f_{i,0} \times 4 \times 2} \\ &= b_1 e^{-j2\pi f_{i,0} \times 4} + b_2 e^{-j2\pi f_{i,0} \times 6} + b_3 e^{-j2\pi f_{i,0} \times 8} \end{aligned}$$

Where $b_1 = a_1^2$, $b_2 = 2a_1a_2$, $b_3 = a_2^2$. Clearly, the above equation has a form similar to a wireless channel with propagation delays 4 ns, 6 ns and 8 ns respectively. This means that applying Chronos's algorithm will result in peaks precisely at 4 ns, 6 ns and 8 ns. Notice that in addition to 4 ns and 8 ns that are simply twice the propagation delays of genuine paths, there is an extra peak at 6 ns. This peak stems from the square operation in $\tilde{h}_{i,0}^2$ and is a sum of two delays. However, the sum of any two delays will always be higher than twice the lowest delay. Consequently, the smallest of these propagation delays is still at 4 ns – i.e., at twice the time-of-flight. A similar argument holds for larger number of signal paths, and can be used to recover time-of-flight.

Finally, we make a few observations: (1) In practice, the forward and reverse channels cannot be measured at exactly the same t but within short time separations (tens of microseconds), resulting in a small phase error. However, this error is significantly smaller than the error from not compensating for frequency offsets altogether (for tens of milliseconds). The error can be resolved by averaging over several packets. (2) Delays in the hardware result in a constant additive value to the time-of-flight. This constant can be pre-calibrated once in the lifetime of a WiFi-card, by measuring time-of-flight to a device at a known distance. (3) Standard Fourier Transform properties dictate that a minimum separation of Δf in frequencies of measured CSI values, leads to an ambiguity by multiples of $\frac{1}{\Delta f}$ in the time estimates (i.e the delay is measured modulo $\frac{1}{\Delta f}$). Since, Chronos uses CSI measurements at center frequencies, the minimum frequency separation is 5 *MHz* [6]. Hence, the time domain ambiguity is 200 *ns* which corresponds to a distance of 60 m, i.e., distance measurements are modulo 60 *m*. Thus, for indoor settings and typical WiFi propagation, one can ignore the modulo factor.

7. Computing Distances and Location

So far, we have explained how Chronos measures the time-of-flight between two antennas on a pair of WiFi cards. One can then compute the distance between the two antennas (i.e., the two devices) by multiplying the time-of-flight by the speed of light.

In order to get the location of the client from the distance measurements, Chronos follows a two-step proce-

[6] The frequency separation is less than the channel bandwidth of 20 MHz due to overlapping WiFi bands.

dure. In the first step, Chronos refines the distance measurements by utilizing geometric constraints, imposed by the relative locations of the antennas on the access point and the client. In the second step, Chronos formulates a quadratic optimization problem, based on the refined distances to get the accurate location of the client with respect to the access point.

Mathematically, we denote the separation between antenna i and antenna j on the access point by l_{ij}^{ap}. Similarly, antenna i and antenna j on the client are separated by $l_{i,j}^{cl}$. By using standard triangle inequality, we know that $|d_{ij} - d_{i'j}| < l_{ii'}^{ap}$, where d_{ij} is the distance measured by Chronos between antenna i on the access point and antenna j on the client. When a pair of distances measured by Chronos violates this constraint; clearly, one or both of the distance measurements must be declared invalid. Chronos uses a relaxed version of triangle inequality to eliminate erroneous distance measurements. Specifically, if we denote the maximum distance between any pair of antennas on a device by α, Chronos chooses the largest cluster, C, of distance measurements such that each measurement in this cluster is at most α away from at least one other distance measurement in the cluster. Chronos, then, discards the distance measurements that do not belong to C.

Finally, Chronos formulates the following constrained optimization problem to find the accurate position of the client. We denote the position of the i^{th} antenna on the access point by (x_i^{ap}, y_i^{ap}). Our goal is to optimize for the position of the client which we denote by (x, y), where x and y are 3×1 vectors of antenna coordinates:

$$\min_{\epsilon>0,x,y} \epsilon$$

such that

$$\forall(i,j) \in C, |dist((x_i^{ap}, y_i^{ap}), (x_j, y_j)) - d_{ij})| < \epsilon$$

$$\forall(i,j) \in \{1,2,3\}, dist((x_i, y_i), (x_j, y_j)) = l_{i,j}^{cl}$$

where $dist((x_1, y_1), (x_2, y_2))$ denotes the euclidean distance between points (x_1, y_1) and (x_2, y_2). On a high level, Chronos optimizes for the minimal violation of the distance constraints while still maintaining the relative position of the antennas on the client. We formulate this problem as a quadratic-constrained optimization in MATLAB and use the fmincon solver to find the optimum solution. The average execution time for this algorithm is 0.09 s (standard deviation 0.01 s).

8. Implementation

We implemented Chronos as a software patch to the iwlwifi driver on Ubuntu Linux running the 3.5.7 kernel. To measure channel-state-information, we use the 802.11 CSI Tool [21] for the Intel 5300 WiFi card. We measure the channels on both 2.4 GHz and 5 GHz WiFi bands.[7]

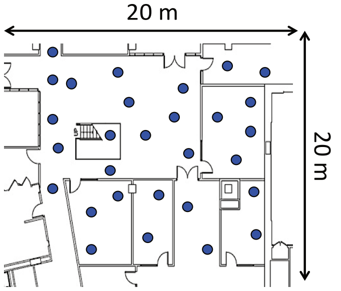


Figure 4: **Lab Testbed:** The figure depicts our testbed with candidate locations for the nodes marked with blue dots.

Unless specified otherwise, we pair two Chronos devices by placing each device in monitor mode with packet injection support on the same WiFi frequency. We implemented Chronos's frequency band hopping protocol (see §3) in the iwlwifi driver using high resolution timers (hrtimers), which can schedule kernel tasks such as packet transmits at microsecond granularity. Since the 802.11 CSI Tool does not report channel state information for Link-Layer ACKs received by the card, we use packet-injection to create and transmit special acknowledgments directly from the iwlwifi driver to minimize delay between packets and acknowledgments. These acknowledgments are also used to signal the next channel that the devices should hop to, as described in §3. We process the CSI to infer time-of-flight and device locations purely in software written in part in C++, MEX and MATLAB.

We note that all our experiments are conducted in naturally dynamic environments, specifically, an office building, a coffee shop and a home with four occupants. Chronos requires no modifications based on the changes in the environment. The environments have ambient WiFi traffic. We could sense 3 to 19 different access points across our testbeds. Chronos disables the contention mechanism during hopping in order to enable fast switching across different WiFi bands. This causes noise in Chronos's measurements when there is a collision with other WiFi packets. However, Chronos is resilient to noise on a small subset of the measurements. Moreover, since Chronos sends few packets on each WiFi band, it does not adversely effect the WiFi traffic.

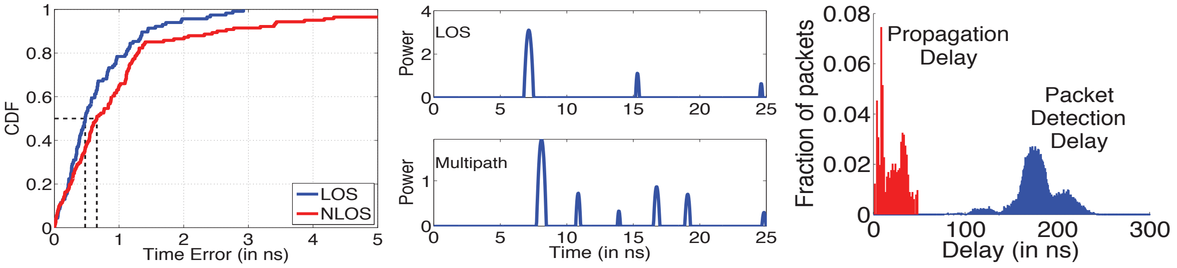
9. Results

We evaluate Chronos's ability to measure the time-of-flight, and compute a client's position using a single AP.

9.1 Time-of-Flight Accuracy

We examine whether Chronos can deliver on its promise

[7] The Intel 5300 WiFi card is known to have a firmware issue on the 2.4 GHz bands that causes it to report the phase of the channel $\angle\tilde{h}_{i,0}$ modulo $\pi/2$ (instead of the phase modulo 2π) [18]. We resolve this issue by performing Chronos's algorithm at 2.4 GHz on $\tilde{h}_{i,0}^4$ instead of $\tilde{h}_{i,0}$. This does not affect the fact that the direct path of the signal will continue being the first peak in the inverse NDFT (like in §6).



(a) Time of Flight (b) Multipath Profiles (c) Packet Detection Delay

Figure 5: **Accuracy in Time of Flight**: (a) The CDF of error in time-of-flight between two devices in Line of Sight (LOS) and Non-Line of Sight (NLOS). (b) Representative multipath profiles. (c) Histograms of time-of-flight and packet detection delay.

of measuring sub-nanosecond time-of-flight between a pair of commodity WiFi devices.

Method: We run our experiments in the testbed in Fig. 4. In each experiment, we randomly pick a location for the AP. We then randomly pick a client location that is within 15 meter from the AP. We experiment with both line-of-sight and non-line-of-sight settings. We perform our experiments using a 10" ASUS EEPC netbook as a client and a Thinkpad W300 Laptop emulating a WiFi AP via hostapd. Both devices are equipped with the 3-antenna Intel 5300 chipset. The antennas are placed at the corner of each device, which results an average antenna spacing of 30cm for the Thinkpad AP and 12cm for the ASUS client.

Using the above setup, we have run 400 localization experiments for different AP-client pairs. For each pair, we run Chronos channel hopping protocol. We compute the time of flight between each transmit antenna and receive antenna. We measure the ground-truth location using a combination of architectural drawings of our building and a Bosch GLM50 laser distance measurement tool [1], which measures distances up to 50 m with an accuracy of 1.5 mm. The ground truth time-of-flight is the ground truth distance divided by the speed of light.

Time-of-Flight Results: We first evaluate Chronos's accuracy in time-of-flight. Fig. 5(a) depicts the CDF of the time-of-flight of the signal in line-of-sight settings and non-line-of-sight. We observe that the median errors in time-of-flight estimation are 0.47 ns and 0.69 ns respectively. These results show that Chronos achieves its promise of computing time-of-flight at sub-nanosecond accuracy. To put this in perspective, consider SourceSync [38], a state-of-the-art system for time synchronization. SourceSync achieves 95th percentile synchronization error up to 20 ns, using advanced software radios. In contrast, the figure shows that Chronos's 95th percentile error is 1.96 ns in line-of-sight and 4.01 ns in non-line-of-sight. Thus, Chronos achieves 5 to 10 fold lower error in time-of-flight, and runs on commodity WiFi cards as opposed to software radios.

Multipath Profile Results: Next, we would like to examine whether multiple path profiles are indeed sparse. Thus, we plot candidate multipath profiles computed by Chronos in the above experiments. Fig. 5(b) plots representative multipath profiles in line-of-sight and multipath environments. We note that both profiles are sparse, with the profile in multipath environments having five dominant peaks. Across all experiments, the mean number of dominant peaks in the multipath profiles is 5.05 on average, with standard deviation 1.95 — indicating that they are indeed sparse. As expected, the profile in line-of-sight has even fewer dominant peaks than the profile in multipath settings. In both cases, we observe that the leftmost peaks in both profiles correspond to the true location of the source. Further, we observe that the peaks in both profiles are sharp due to two reasons: 1) Chronos effectively spans a large bandwidth that includes all WiFi frequency bands, leading to high time resolution; 2) Chronos's resolution is further improved by exploiting sparsity that focuses on retrieving the sparse dominant peaks at much higher resolution, as opposed to all peaks.

Packet Detection Delay Results: Past work on WiFi time measurement and/or synchronization cannot measure the time-of-flight of a packet separately from its detection delay [38]. ([35] measures the distribution of detection delays but not the detection delay of a particular packet.) In contrast, Chronos has a novel way for separating the detection delay from the time-of-flight. We would like to understand the importance of this capability for the success of Chronos. Thus, we use the measurements from the above experiments to compare time-of-flight in indoor environments against packet detection delay.

Fig. 5(c) depicts histograms of both packet detection delay and time-of-flight across experiments. Chronos observes a median packet detection delay of 177 ns across experiments. We emphasize two key observations: (1) Packet detection delay is nearly 8× larger than the time-of-flight in our typical indoor testbed. (2) Packet delay varies dramatically between packets, and has a high standard deviation of 24.8 ns. In other words, packet detection delays are large, highly variable, and hard to predict. This means that if left uncompensated, these delays could lead to a large error in time-of-flight measurements. Our results therefore reinforce the importance of accounting for these delays and demonstrate Chronos's ability to do so.

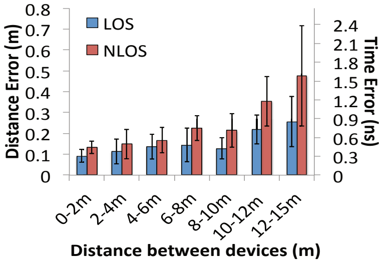


Figure 6: **Ranging Accuracy:** Plots error in distance across the true distance separating the transmitter from the receiver.

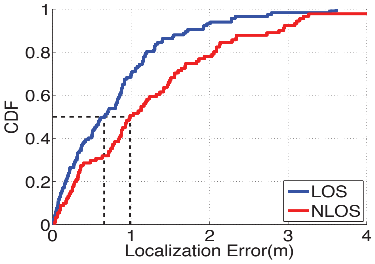


Figure 7: **Localization Accuracy:** Plots CDF of localization error in Line-of-Sight (LOS) and Non-Line-of-Sight (NLOS).

9.2 Localization Accuracy

We evaluate Chronos's accuracy in measuring distance and location using a single access point.

Method: We compute the time-of-flight between the AP and user client in the testbed as described in §9.1 above. We use the measured time-of-flight to compute the distance between antennas and localize the client with respect to the AP as described in §7. We repeat the experiment multiple times in line-of-sight and non-line-of-sight.

Location Results: Fig. 7 plots a CDF of localization error using Chronos in different settings. The device's median positioning error is 65 cm and 98 cm in line-of-sight and non-line-of-sight respectively. This result shows that Chronos's accuracy is comparable to state-of-the-art indoor localization that use multiple AP's [30, 32, 48].

Ranging Results: In some applications, it is important to maintain a particular distance between objects but the exact location is not necessary (e.g., preventing robot collision). Thus, here we plot the ranging results of Chronos. Fig. 6 plots the median and standard deviation of error in distance computed between the transmitter and receiver against their true distance. We observe that this error is initially around 10 cm and increases to at most 26 cm at 12-15 meters. The increase is primarily due to reduced signal-to-noise ratio at further distances. Note that the ranging accuracy is higher than the localization accuracy because ranging is a simpler problem (no need to find the exact direction) and Chronos's time-of-flight computation naturally yields the range between devices.

9.3 Impact on Network traffic

Chronos enables localization between a pair of WiFi devices without third party support. In many cases, these are user devices that do not otherwise communicate data between each other directly. However, an interesting question is the impact of Chronos on network traffic, if one of the devices is serving traffic, such as a WiFi AP. This experiment answers three questions in this regard: (1) How long does Chronos take to hop between all WiFi bands? (2) How does Chronos impact real-time traffic like video streaming applications? (3) How does Chronos affect TCP? We address these questions below:

Method: We consider a Thinkpad W530 Laptop emulating an AP and two ASUS EEPC netbook clients. We assume client-2 requests the AP for indoor localization at $t = 6$ s. We measure the time Chronos incurs to hop between the 35 WiFi bands. Meanwhile, client-1 runs a long-lasting traffic flow. We consider two types of flows: (1) VLC video stream over RTP; (2) TCP flow using iperf. We run the experiment 30 times and find aggregate results.

Results: Fig. 8(a) depicts the CDF of the time that Chronos incurs to hop over all WiFi bands. We observe that the median hopping time is 84 ms for the Intel 5300 WiFi card, like past work on commercial WiFi radios [29].

Next, Fig. 8(b) plots a representative trace of the cumulative bytes of video received over time of a VLC video stream run by client-1 (solid blue line). The red line plots the cumulative number of bytes of video played by the client. Notice that at $t = 6$ s, there is a brief time span when no new bytes are downloaded by the client (owing to the localization request). However, in this interval, the buffer has enough bytes of video to play, ensuring that the user does not perceive a video stall (i.e. the blue and red lines do not cross). In other words, buffers in today's video streaming applications can largely cushion such short-lived outages [26, 25], minimizing impact on user experience. Similarly, Fig. 8(c) depicts a representative trace of the throughput over time of a TCP flow at client-1. The TCP throughput dips only slightly by 18.5% at $t = 6$ s, when client-2 requests location. Thus, Chronos can support localization without much impact on data traffic. However, if more frequent localization is desired with large traffic demands, we recommend deploying a dedicated AP or WiFi beacon for localization.

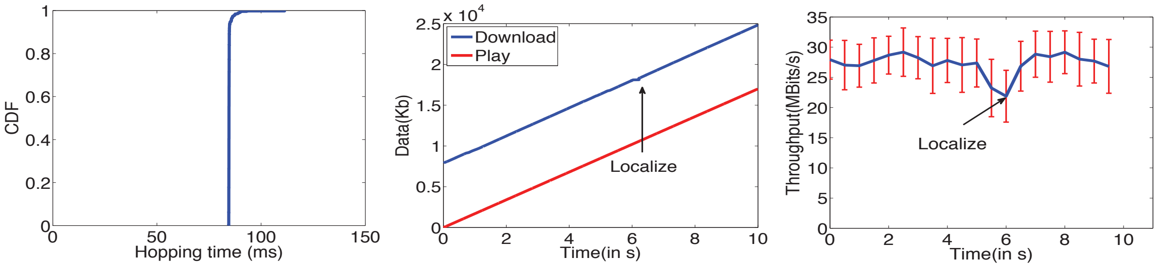
10. Applications

We evaluate Chronos in three application scenarios.

10.1 Room Occupancy Detection

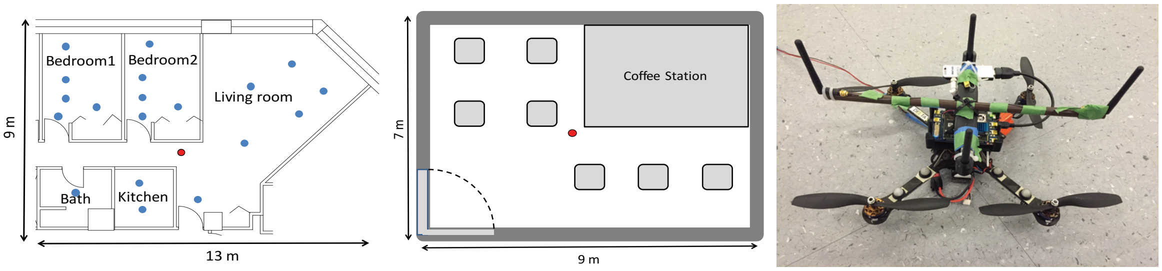
Smart home technologies, such as personalized heating and lighting, can vastly benefit from information about the number and identity of people in individual rooms. Chronos is a natural solution for this problem as it can localize and track people using their smartphones and wearables, even if the home has a single WiFi access point.

Method: To demonstrate this capability, we deployed Chronos in a two-bedroom apartment that has four res-



(a) Hopping Time (b) Video Streaming (c) TCP Throughput

Figure 8: **Impact on Network Traffic:** (a) measures the CDF of time taken by Chronos to hop between all WiFi bands – a small value of 84 ms. Consider a client-1 with a long-running traffic flow to an AP. The AP is asked to localize another client-2 at $t = 6$ s. (b) depicts a representative trace of the number of bytes of data downloaded and data played over time if the client-1 views a VLC video stream. (c) measures the throughput if client-1 runs a TCP flow using iperf. In either case, the impact of client-1's flow is minimal at $t = 6$ s.



(a) Home Floor Plan (b) Coffee Shop Schematic (c) Personal Drone

Figure 9: (a) Floor map of the apartment where Chronos is deployed. Red dot indicates the access point and the blue dots represent the client positions. (b) Coffee shop schematic. Red dot indicates the access point. (c) We implement Chronos on an AscTec Hummingbird quadrotor with an AscTec Atomboard.

idents. The floor map of the apartment is shown in Fig. 9(a). The Chronos access point is centrally placed in the home and is indicated by the red dot. Each resident is given an ASUS netbook, equipped with Intel 5300 WiFi cards, and running Chronos. The residents are then asked to move freely to locations within the apartment. Their locations are manually recorded and are marked by the blue dots in Fig. 9(a). Chronos measures the location of each resident and detects the room the person is in. In particular, Chronos distinguishes between the two bed rooms, living room, kitchen and bathroom.

Results: In our experiments, Chronos detects the user to be in the correct room in 94.3% of the experiments. Most of the errors occurred in Bedroom 1 in Fig. 9(a), and were due to the signal being too weak after traversing two walls and a closet. Overall, the results show that Chronos can enable applications based on room occupancy detection with a single home access point.

10.2 WiFi Geo-Fencing

Chronos can be used to authorize WiFi access in small businesses, which have only one access point. To demonstrate this capability, we deploy Chronos in a popular coffee shop with free WiFi, and use the distance from the access point to measure whether an individual is inside or outside the coffee shop (Fig. 9(b)).

Method: We conducted 100 experiments in the coffee shop. The user used an ASUS netbook, equipped with the Intel 5300 WiFi card to connect to the Chronos AP. In 50 of these experiments, the user was standing at a randomly chosen location inside the coffee shop, while in the others, the user was standing outside, while still being able to access the WiFi connection.

Results: Chronos correctly inferred whether the user was inside or outside in 97% of experiments. However, if we simply authenticate users based on location without any buffer zone, the accuracy is 97%, but one legitimate customer cannot access WiFi in his current location. In contrast, if we decide to accept users located within 30 cm of the premises, Chronos authenticates all legitimate customers but allows access incorrectly to people outside the premise in 5% of the experiments, decreasing the overall accuracy to 95%. Since it is more important to ensure customers can access WiFi, we believe that one should use some buffer zone.

10.3 Personal Drones

We apply Chronos to indoor personal drones [11]. These drones can follow users around while maintaining a convenient distance relative to the control device in the user's hand or pocket. Users can use these drones to take pictures or videos of them while performing an activity, even in indoor settings where GPS is unavailable.

Method: We use an AscTec Hummingbird quadrotor

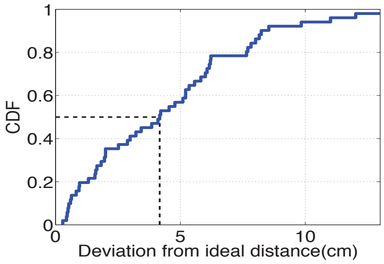


Figure 10: **Application to Personal Drones:** The drone uses Chronos to maintain a constant distance of 1.4 m to the user. The figure plots the CDF of errors in maintaining a distance of 1.4 m.

equipped with the AscTec Atomboard light-weight computing platform (with the Intel 5300 WiFi card), a Go-pro camera and a Yei-Technology motion sensor. Fig. 9(c) depicts our setup. Note that the Intel 5300 WiFi card supports 3-antennas; the fourth antenna on the quadrotor is placed only for balance and stability.

We perform our personal drone experiments in a 6 m × 5 m room augmented with the VICON motion capture system [2]. We use VICON to find the ground-truth trajectories of the personal drone and the user control device. In each experiment, the personal drone tracks an ASUS EEPC netbook with the Intel 5300 WiFi card held by a user. The distance measurements from Chronos are integrated with drone navigation using a standard negative feedback-loop robotic controller [12]. The drone maintains a constant height and follows the user to maintain a constant distance of 1.4 m relative to the user's device. The drone also captures photographs of the user along the way using the Go-Pro camera mounted on the Hummingbird quadrotor, keeping the user at 1.4 m in focus. The drone uses the compass on the user's device and the quadrotor to ensure that its camera always faces the user.

Results: Fig. 10 measures the CDF of root mean squared deviation in distance of the drone relative to the desired value of 1.4 m — a median of 4.17 cm. Our results reveal that the drone tightly maintains its relative distance to the user's device. Notice that our error in distance is significantly lower in this experiment relative to §9.2. This is because drones measure multiple distances as they navigate in the air, which helps de-noise measurements and remove outliers. Chronos is the first system to achieve such a high accuracy in device to device positioning using no support from surrounding infrastructure.

11. Related Work

Chronos builds on vast literature on indoor WiFi-based localization [40, 13, 9, 51, 18, 48, 32, 30, 50, 4]. However, past work that delivers sub-meter location accuracy typically requires cooperation across multiple (four or five) AP's [18, 48, 32, 30, 50].

A few prior proposals have aimed to localize with a single WiFi AP. They may be divided into two categories: some proposals [31, 52] require exhaustive fingerprinting of received signal power prior to deployment. Such proposals exhibit localization errors of several meters and incur a large overhead due to fingerprinting. The second class of proposals attempt to measure time-of-flight either directly [35], or indirectly using the phase [53]. However, since they cannot accurately measure the time-of-flight, they need the user to walk around, perform measurements in multiple locations, and intersect those measurements with the help of an accelerometer. In contrast, Chronos has tens of centimeter accuracy, and neither requires fingerprinting nor user motion.

A few past papers on WiFi-based localization leverage channel hopping [50, 49]. However, unlike Chronos which measures the absolute time-of-flight and localizes with a single AP, those systems measure differences in the time-of-flight and require the deployment of multiple AP's.

Prior theoretical ranging algorithms [44, 43] have used the Chinese Remainder theorem. However, Chronos differs from those algorithms in multiple ways. First, those algorithms ignore multipath and assume that wireless signals propagate in free space with a single time-of-flight value. In contrast, Chronos addresses the crucial problem of multipath, and hence its complete algorithm uses non-uniform Fourier transform as opposed to the Chinese Remainder theorem. Second, those algorithms ignore pratical issues such as the frequency offset between the transmitter and the receiver and the inability of the receiver to separate the time of flight from the packet detection delay.

Finally, some past work has explored measuring the time-of-flight of WiFi signals for reasons other than localization. There have been several studies that resolve time-of-flight to around ten nanoseconds using the clocks of WiFi cards or other methods [46, 33, 17, 34, 38]. In contrast, Chronos can achieve sub-nanosecond resolution which is necessary for accurate localization.

12. Conclusion

This paper presents Chronos, a system that measures sub-nanosecond time-of-flight on commercial WiFi radios. Chronos uses these measurements to enable WiFi device-to-device positioning at state-of-the-art accuracy, without support of additional WiFi infrastructure or non-WiFi sensors. By doing so, Chronos opens up WiFi-based positioning to new applications where additional infrastructure and sensors may be unavailable or inaccessible, e.g., geo-fencing, home occupancy measurements, finding lost devices, maintaining robotic formations, etc.

Acknowledgements: We thank the NETMIT group, Arthur Berger, our reviewers and our shepherd, Alex Snoeren, for their insightful comments. This work is funded by NSF. We thank members of the MIT Center for Wireless Networks and Mobile Computing: Amazon, Cisco, Google, Intel, Mediatek, Microsoft, ST Microelectronics and Telefonica for their interest and support.

13. References

[1] Bosch Laser Distance Measurer GLM50. http://www.boschtools.com/Products/Tools/Pages/BoschProductDetail.aspx?pid=GLM\%2050.

[2] VICON T-Series. http://www.vicon.com/products/documents/Tseries.pdf.

[3] IEEE 802.11n-2009 Standard. 2009. http://standards.ieee.org/findstds/standard/802.11n-2009.html.

[4] O. Abari, D. Vasisht, and D. Katabi. Caraoke: An E-Toll Transponder Network for Smart Cities. SIGCOMM, 2015.

[5] F. Adib, Z. Kabelac, D. Katabi, and R. C. Miller. 3D Tracking via Body Radio Reflections. NSDI, 2014.

[6] Automated Home. Apple iBeacons Explained: Smart Home Occupancy Sensing Solved?, 2013.

[7] F. Bach, R. Jenatton, J. Mairal, and G. Obozinski. Convex Optimization with Sparsity-Inducing Norms, 2011.

[8] S. Bagchi and S. K. Mitra. *The Nonuniform Discrete Fourier Transform and Its Applications in Signal Processing*. 1999.

[9] P. Bahl and V. Padmanabhan. RADAR: An in-building RF-based User Location and Tracking System . INFOCOM, 2000.

[10] W. U. Bajwa, J. Haupt, A. Sayeed, and R. Nowak. Compressed Channel Sensing: A New approach to Estimating Sparse Multipath Channels. Proceedings of the IEEE, 2010.

[11] Ben Popper. The Drone You Should Buy Right Now, 2014. http://www.theverge.com/2014/7/31/5954891/best-drone-you-can-buy.

[12] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. *Software Engineering for Self-Adaptive Systems*. 2009.

[13] K. Chintalapudi, A. Padmanabha Iyer, and V. N. Padmanabhan. Indoor Localization without the Pain. MobiCom, 2010.

[14] C. Ding, D. Pei, and A. Salomaa. *Chinese Remainder Theorem: Applications in Computing, Coding, Cryptography*. 1996.

[15] A. Dutt and V. Rokhlin. Fast Fourier Transforms for Nonequispaced Data. *SIAM J. Sci. Comput.*, 1993.

[16] S. Gezici, Z. Tian, G. B. Biannakis, H. Kobayashi, A. F. Molisch, V. Poor, Z. Sahinoglu, S. Gezici, Z. Tian, G. B. Giannakis, H. Kobayashi, A. F. Molisch, H. V. Poor, and Z. Sahinoglu. Localization via Ultra-wideband Radios. In *IEEE Signal Processing Magazine*, 2005.

[17] D. Giustiniano and S. Mangold. CAESAR: Carrier Sense-based Ranging in Off-the-shelf 802.11 Wireless LAN. CoNEXT, 2011.

[18] J. Gjengset, J. Xiong, G. McPhillips, and K. Jamieson. Phaser: Enabling Phased Array Signal Processing on Commodity WiFi Access Points. MobiCom, 2014.

[19] L. Greengard and J. Lee. Accelerating the Nonuniform Fast Fourier Transform. *SIAM REVIEW*, 2004.

[20] M. Guillaud, D. Slock, and R. Knopp. A Practical Method for Wireless Channel Reciprocity Exploitation through Relative Calibration. 2005.

[21] D. Halperin, W. Hu, A. Sheth, and D. Wetherall. Tool Release: Gathering 802.11n Traces with Channel State Information. ACM SIGCOMM CCR, 2011.

[22] J. Heiskala and J. Terry. *OFDM Wireless LANs: A Theoretical and Practical Guide*. Sams Publishing, 2001.

[23] B. Hofmann-Wellenhof, H. Lichtenegger, and J. Collins. *Global Positioning System: Theory and Practice*. Springer Science & Business Media, 2013.

[24] K. Hou, Z. Zhou, A. M.-C. So, and Z.-Q. Luo. On the Linear Convergence of the Proximal Gradient Method for Trace Norm Regularization. NIPS, 2013.

[25] T.-Y. Huang, R. Johari, and N. McKeown. Downton Abbey Without the Hiccups: Buffer-based Rate Adaptation for HTTP Video Streaming. FhMN, 2013.

[26] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. SIGCOMM, 2014.

[27] A. T. Islam and I. Misra. Performance of Wireless OFDM System with LS-Interpolation-Based Channel Estimation in Multi-path Fading Channel. IJCSA, 2012.

[28] K. Joshi, S. Hong, and S. Katti. PinPoint: Localizing Interfering Radios. NSDI, 2013.

[29] S. Kandula, K. C.-J. Lin, T. Badirkhanli, and D. Katabi. FatVAP: Aggregating AP Backhaul Capacity to Maximize Throughput. NSDI, 2008.

[30] M. Kotaru, K. Joshi, D. Bharadia, and S. Katti. SpotFi: Decimeter Level Localization Using WiFi. SIGCOMM, 2015.

[31] C. Kumar, R. Poovaiah, A. Sen, and P. Ganadas. Single Access Point-based Indoor Localization Technique for Augmented Reality Gaming for Children. Students' Technology Symposium (TechSym), 2014 IEEE, 2014.

[32] S. Kumar, S. Gil, D. Katabi, and D. Rus. Accurate Indoor Localization with Zero Start-up Cost. MobiCom, 2014.

[33] S. Lanzisera, D. Zats, and K. Pister. Radio Frequency Time-of-Flight Distance Measurement for Low-Cost Wireless Sensor Localization. *Sensors*

Journal, IEEE, 2011.

[34] A. Marcaletti, M. Rea, D. Giustiniano, V. Lenders, and A. Fakhreddine. Filtering Noisy 802.11 Time-of-Flight Ranging Measurements. CoNEXT, 2014.

[35] A. T. Mariakakis, S. Sen, J. Lee, and K.-H. Kim. SAIL: Single Access Point-based Indoor Localization. MobiSys, 2014.

[36] A. Nag and S. Mukhopadhyay. Occupancy Detection at Smart Home Using Real-Time Dynamic Thresholding of Flexiforce Sensor. *Sensors Journal, IEEE*, 2015.

[37] New York Times. Homes Try to Reach Smart Switch, 2015. `http://www.nytimes.com/2015/04/23/business/energy-environment/homes-try-to-reach-smart-switch.html`.

[38] H. Rahul, H. Hassanieh, and D. Katabi. SourceSync: A Distributed Wireless Architecture for Exploiting Sender Diversity. ACM SIGCOMM, 2010.

[39] H. Rahul, S. Kumar, and D. Katabi. MegaMIMO: Scaling Wireless Capacity with User Demands. ACM SIGCOMM, 2012.

[40] A. Rai, K. K. Chintalapudi, V. N. Padmanabhan, and R. Sen. Zee: Zero-effort Crowdsourcing for Indoor Localization. MobiCom, 2012.

[41] P. Setlur, G. Alli, and L. Nuzzo. Multipath Exploitation in Through-Wall Radar Imaging Via Point Spread Functions. *IEEE Transactions on Image Processing*, 2013.

[42] D. Tse and P. Vishwanath. *Fundamentals of Wireless Communications*. Cambridge University Press, 2005.

[43] C. Wang, Q. Yin, and H. Chen. Robust Chinese Remainder Theorem Ranging-method based on Dual-Frequency Measurements. *IEEE Transactions on Vehicular Technology*, 2011.

[44] C. Wang, Q. Yin, and W. Wang. An Efficient Ranging Method for Wireless Sensor Networks. ICASSP, 2010.

[45] E. Weisstein. Chinese Remainder Theorem. `http://mathworld.wolfram.com/ChineseRemainderTheorem.html`.

[46] S. B. Wibowo, M. Klepal, and D. Pesch. Time of Flight Ranging using Off-the-self IEEE802.11 WiFi Tags. POCA, 2009.

[47] WiFi Alliance. Make Security a Priority in 2011: Protect Your Personal Data on Wi-Fi Networks, 2011. `http://www.wi-fi.org/news-events/newsroom/make-security-a-priority-in-2011-protect-your-personal-data-on-wi-fi-networks`.

[48] J. Xiong and K. Jamieson. ArrayTrack: A Fine-Grained Indoor Location System. NSDI, 2013.

[49] J. Xiong, K. Jamieson, and K. Sundaresan. Synchronicity: Pushing the Envelope of Fine-grained Localization with Distributed Mimo. HotWireless, 2014.

[50] J. Xiong, K. Sundaresan, and K. Jamieson. ToneTrack: Leveraging Frequency-Agile Radios for Time-Based Indoor Wireless Localization. MobiCom, 2015.

[51] M. Youssef and A. Agrawala. The Horus WLAN Location Determination System. MobiSys, 2005.

[52] G. V. Zàruba, M. Huber, F. Kamangar, and I. Chlamtac. Indoor Location Tracking using RSSI Readings From a Single Wi-Fi Access Point. *Wireless networks*, 2007.

[53] X. Zheng, C. Wang, Y. Chen, and J. Yang. Accurate Rogue Access Point Localization Leveraging Fine-grained Channel Information. IEEE Conference on Communications and Network Security (CNS), 2014.

A Scalable Multi-User Uplink for Wi-Fi

Adriana B. Flores, Sadia Quadri and Edward W. Knightly
Department of Electrical and Computer Engineering, Rice University, Houston, TX
{adriana.flores, sadia.quadri, knightly}@rice.edu

Abstract

Mobile devices have fewer antennas than APs due to size and energy constraints. This antenna asymmetry restricts uplink capacity to the *client* antenna array size rather than the AP's. To overcome antenna asymmetry, multiple clients can be grouped into a simultaneous multi-user transmission to achieve a full rank transmission that matches the number of antennas at the AP. In this paper, we design, implement, and experimentally evaluate MUSE, the first distributed and scalable system to achieve full-rank uplink multi-user capacity *without* control signaling for channel estimation, channel reporting, or user selection. Our experiments demonstrate full-rank multiplexing gains in the evaluated scenarios that show linear gains as the number of users increase while maintaining constant overhead.

1 Introduction

Form factor and energy constraints result in mobile clients having significantly fewer antennas than access points (APs), e.g., 1 or 2 for clients vs. 8 or even more for massive MIMO APs [1, 2]. When an AP serves a single user at a time, this asymmetry severely restricts capacity with limits defined by the *client* antenna array size rather than the AP's. Fortunately, both theoretical results and practical implementations have shown that multiple clients can be grouped into a simultaneous multi-user (MU-MIMO) transmission. The transmission can achieve "full rank" when the group of clients form a virtual array having the same number of antennas as the AP [1, 2, 3, 4, 5]. While this technique is already standardized [6, 7] and commercialized [8, 9] for the Wi-Fi downlink, the uplink still serves a single user at a time, as specified in the original 1997 Wi-Fi standard [10].

In this paper, we design, implement, and experimentally evaluate MUSE, the first system to achieve full-rank uplink multi-user capacity *without* requiring a control channel.[1] Namely, mirroring the functionality of downlink standards for the uplink would require a control channel (set of control message exchanges, etc.) for mechanisms such as channel estimation, reporting of channel state, joint stream precoding, orthogonal user selection, and control of the timing of user transmissions. In contrast, we develop MUSE to scale not only raw physical layer capacity, but also system throughput after incorporating all protocol overhead. In particular, we present the following contributions.

First, we design three physical layer components to enable the aforementioned MUSE properties (MUSE-PHY). *(i)* In order for multiple users to transmit simultaneously, their combined transmissions must be sufficiently orthogonal to be successfully decoded by the AP. Rather than measuring channels and performing user selection, indoor multipath induces sufficient channel independence for an arbitrary group of users to transmit concurrently. However, precise channel estimation is required at the receiver, the AP must estimate the *joint* channel state in order to separate and decode the streams. We introduce the Dynamic Orthogonal Mapping (DOM) matrix as a mechanism compatible with the IEEE 802.11 standard, that allows the AP to obtain clean channel estimations from all independent distributed transmitters. We redesign the 802.11n SU-MIMO (Single User MIMO) channel estimation to function with multiple distributed users. Contrary to the requirement that multiplexing capabilities of SU-MIMO be pre-configured at the transmitter, in MUSE, DOM is dynamically matched according to the transmission and only statically limited by the receiver (AP). DOM preamble-based channel training avoids the non-scalable approach of sequentially training one user at a time. *(ii)* DOM channel training requires the same symbols to be transmitted by all the distributed stations and can result in signal correlation and unintended beamforming. Exploiting cyclic de-

[1] MUSE is an acronym for M̲ulti-U̲ser S̲calablE̲ Uplink.

lay diversity to increase diversity in the signal paths [3], we introduce Arbitrary Cyclic Shift Delay (aCSD) to maximally decorrelate users' signals at the transmitters. Recognizing that there is no control channel among the transmitters and leveraging their physical separation, we employ arbitrary cyclic shift delays applied independently by each transmitter and demonstrate that these achieve the desired multiplexing gains in a distributed and non-channel-dependent manner. *(iii)* Lastly, because MUSE lacks a control channel, the number of uplink data streams is not fixed before the transmission. Consequently, the MUSE design supports a variable number of arbitrarily selected users and provides flexibility and robustness to unknown client backlog state via dynamic use of the DOM matrix.

Second, we design a medium access control protocol (MUSE-MAC) that exploits the unique MUSE physical-layer capabilities. Namely, *without* MUSE-PHY, the MAC design would require a control protocol with mechanisms described above such as: feedback of channel state, selection of an orthogonal group of transmitters, alignment of their timing, and elimination of uncertainty in the number of transmitting users. In contrast, we exploit MUSE-PHY's ability to support an arbitrary set of users and design MUSE-MAC to select a *group* of arbitrary users with the rank of the group not exceeding the number of antennas at the AP. In principle, a random group could be selected by invoking the existing Wi-Fi random access technique multiple times in sequence, once to select each user, with the process ending when the rank limit is reached. Unfortunately, such a procedure would require control overhead (in the form of messages and backoff delays) that increases linearly with rank. In contrast, we select a *group* of users with a *single* contention in which all backlogged users independently contend for the channel using the same Wi-Fi random access count-down procedure. When a *single* user wins the channel, MUSE-MAC attaches a random set of additional users to the winning user via a predetermined operation that is a function of each user's Wi-Fi standard Association ID. The groups are consequently *arbitrary* (emulating randomly selected) but predefined, since association IDs are predefined. In this way, we eliminate the need for a group-selection control procedure for each transmission as the MUSE-PHY ensures that a random set of users can be decoded. Even though Association ID grouping may be considered fixed, MUSE enables group adaptation through the reassignment of Association IDs by utilizing 802.11 standardized Reassociation Request and Response procedure.

Finally, we implement MUSE on a software-defined-radio platform, create a WLAN testbed, and evaluate performance using extensive over-the-air experiments. We demonstrate full rank multiplexing gains by orthogonal spatial spreading the distributed transmitters and maintaining a constant overhead as the number of users increases. Specifically, MUSE achieves on average 197%, 290% and 395% aggregate PHY throughput for 2 to 4 concurrent served users respectively. We find that aCSD enables the distributed transmitters to effectively induce multipath in the form of variable phase offset, which results in accurate channel estimation using the DOM matrix. Further, we evaluate the effectiveness of random user grouping and find that while the vast majority of user groupings yield full rank, ill-conditioned channels can occur, necessitating reduction of modulation and coding rate to counter interference. Lastly, we evaluate medium access scalability and demonstrate that as the number of users increases, MUSE MAC-layer throughput, incorporating overhead, scales linearly. MUSE achieves 2.5x higher throughput for a 16 antenna system compared to prior multi-user uplink schemes [11, 12].

2 Orthogonal Multi-User Uplink PHY

In this section, we introduce background in channel estimation for multi-user transmissions and present MUSE's key PHY techniques that enable interference-free channel estimation, decorrelate users' channels, and enable arbitrary user selection.

2.1 Background on CSI

Channel State Information (CSI) at the transmitter (CSIT) or receiver (CSIR) is necessary for multi-stream communication, i.e., for simultaneously spatially multiplexing independent data streams.

CSIT: One method for multi-stream transmission is transmitter-based precoding in order to nullify or zero-force the inter-stream interference, e.g., [4]. Transmit beamforming requires CSIT which is obtained via a closed-loop process in protocols such as IEEE 802.11ac [6, 7]. As shown in Figure 1, a closed-loop approach uses receiver feedback of the estimated CSI. The transmitter then uses CSIT-based signal precoding to uncorrelate users' channels and achieve stream orthogonality with reduced inter-stream interference.[2] This process of collecting CSIT, also termed channel sounding, requires exchange of control information that scales linearly with the number of users, thereby decreasing throughput proportionately due to the resulting air-time cost of control overhead.

CSIR: An open-loop receiver based approach is illustrated in Figure 1, performs CSI acquisition at the receiver at the time of packet transmission. Such CSIR

[2] Likewise, CSIT can be obtained via implicit feedback in which the receiver sends pilots and the transmitter assumes that channels are reciprocal.

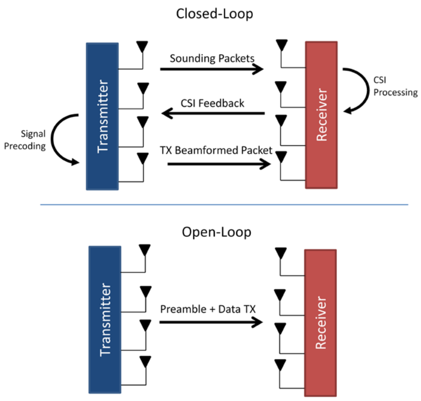


Figure 1: Closed-Loop and Open-Loop Channel State Information (CSI) acquisition and utilization.

estimation is performed through predefined preamble sequences, enabling the receiver to compute the unknown channel given the known preamble data. In an open-loop system, transmitters do not have CSIT. The key benefit of use of CSIR is elimination of control overhead for CSI feedback. However, the main drawback of an open-loop approach is that transmitted streams could have correlated channels yielding inter-stream interference.

2.2 Dynamic Orthogonal Mapping matrix

MUSE open-loop design must address inter-stream interference and correlated channels among concurrent users. To obtain precise CSIR and decode the multi-user transmit data, the estimated CSIR must contain the combination of all transmit signals.

We design channel training to be compatible with the IEEE 802.11 standard where we expand the usage of the 802.11n SU-MIMO channel estimation to function with multiple distributed users. We present the first generalization of the preamble-based channel training of 802.11n to be used by distributed transmitters to achieve multi-user spatial multiplexing gains.

To illustrate MUSE CSIR estimation, consider a 2x2 multi-user uplink transmission in which two clients concurrently communicate with a two antenna AP as shown in Figure 2, which depicts the four channels between the clients and AP.

Following the procedure of 802.11n channel training, the preamble training signals must be transmitted at the same time. While in 802.11n this is easily achievable due to having a single transmitter, in MUSE we expand channel estimation to multiple users. Our MAC design (Section 3) ensures the training signals are sent simultaneously. In effect, receiver antenna Y_1 receives combined high throughput preamble, HTLTF, of both transmitters X_1 and X_2. Equation (1) shows the frequency domain representation of the signals received by the AP for a single subcarrier sc. y_1^{sc} and y_2^{sc} represent the received signals for antenna 1 and 2 respectively and $h_{rx,tx}$ represent the channel taps for a given receiver-transmitter antenna combination.

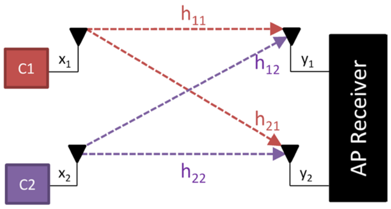


Figure 2: 2x2 Uplink Multi-User MIMO.

$$\begin{aligned} y_1^{sc} &= \hat{h}_{11}^{sc} \cdot HTLTF + \hat{h}_{12}^{sc} \cdot HTLTF + z_1^{sc} \\ y_2^{sc} &= \hat{h}_{21}^{sc} \cdot HTLTF + \hat{h}_{22}^{sc} \cdot HTLTF + z_2^{sc} \end{aligned} \tag{1}$$

As observed we have four unknown channels $(\hat{h}_{11}, \hat{h}_{12}, \hat{h}_{21}, \hat{h}_{22})$ and only two receive antenna equations (Y_1, Y_2). Consequently to be able to resolve all four unknown channels we require four preamble transmissions with a specific linear combination that allows estimation of each channel. The 802.11n standard adds a second set of preamble transmissions with a corresponding linear combination to allow the derivation of all unknown channels as shown in Equation (2). However, when we expand this functionality to multiple-users without a control channel to coordinate the distributed users, these users are required to know the number of preamble signals to send and the linear combination that enables the channel estimation.

$$\begin{aligned} y_{1,t_1}^{sc} &= \hat{h}_{11}^{sc} \cdot HTLTF + \hat{h}_{12}^{sc} \cdot HTLTF + z_{1,t1}^{sc} \\ y_{1,t_2}^{sc} &= -\hat{h}_{11}^{sc} \cdot HTLTF + \hat{h}_{12}^{sc} \cdot HTLTF + z_{1,t2}^{sc} \\ \\ y_{2,t_1}^{sc} &= \hat{h}_{21}^{sc} \cdot HTLTF + \hat{h}_{22}^{sc} \cdot HTLTF + z_{2,t1}^{sc} \\ y_{2,t_2}^{sc} &= -\hat{h}_{21}^{sc} \cdot HTLTF + \hat{h}_{22}^{sc} \cdot HTLTF + z_{2,t2}^{sc} \end{aligned} \tag{2}$$

Consequently, we introduce the DOM matrix which represents the full-rank version of the 802.11n orthogonal mapping matrix and is made available to all devices irrespective of their number of RF chains:

$$DOM = \begin{pmatrix} 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \\ -1 & 1 & 1 & 1 \end{pmatrix}. \tag{3}$$

The dimension of the DOM matrix is dependent on the number of spatial streams (N_{SS}) and the number of HTLTF (N_{HTLTF}) transmitted. While the size of the 11n-standard orthogonal mapping matrix depends on the available RF chains of the *transmitter*, in DOM, the matrix size is fixed to the AP's full-rank version of the 802.11n matrix, yet it operates dynamically as follows.

Through MUSE-MAC design, the distributed users obtain the total number of spatial streams in the multi-user transmission and the assigned Stream ID. With this information and the DOM matrix, the distributed users transmit the required number of HTLTF symbols with the appropriate precoding. Specifically, the transmitter defines the size of the DOM matrix by N_{SS} and applies the row of the DOM matrix corresponding to the Stream ID to the training signals.

Consequently, when all users transmit concurrently the full DOM matrix is formed which permits a receiver to derive all channels by adding or subtracting the HTLTF symbols.

$$\begin{aligned} \hat{h}^{sc}_{11} &= \frac{y^{sc}_{1,t_1} - y^{sc}_{1,t_2}}{2 \cdot HTLTF} & \hat{h}^{sc}_{12} &= \frac{y^{sc}_{1,t_1} + y^{sc}_{1,t_2}}{2 \cdot HTLTF} \\ \hat{h}^{sc}_{21} &= \frac{y^{sc}_{2,t_1} - y^{sc}_{2,t_2}}{2 \cdot HTLTF} & \hat{h}^{sc}_{22} &= \frac{y^{sc}_{2,t_1} + y^{sc}_{2,t_2}}{2 \cdot HTLTF} \end{aligned} \tag{4}$$

Equation (4) shows how all four channels of the example in Figure 2 are estimated by combining received subcarriers from symbol one ($y^{sc}_{t_1}$) with subcarriers from symbol two ($y^{sc}_{t_2}$) and normalized by the number of HTLTFs transmitted. Specifically, combining symbols (t_1 and t_2) from antenna 1 derive channel estimates $\hat{H}^{sc}_{11}$ and $\hat{H}^{sc}_{12}$ and symbols for antenna 2 derive channel estimates $\hat{h}^{sc}_{21}$ and $\hat{h}^{sc}_{22}$. For example as seen in Equation (4), to derive channel estimate $\hat{h}^{sc}_{11}$, the first symbol of antenna 1 (y^{sc}_{1,t_1}) is subtracted by the second symbol (y^{sc}_{1,t_2}) eliminating the $\hat{h}^{sc}_{12}$ term. Then adding symbol one with symbol two of antenna 1 eliminates $\hat{h}^{sc}_{11}$ and solves for $\hat{h}^{sc}_{12}$. This same derivation applies for antenna two to obtain $\hat{h}^{sc}_{21}$ and $\hat{h}^{sc}_{22}$.

2.3 Arbitrary Cyclic Shift Delay

MUSE achieves interference-free channel estimation through linear combination of preambles with the DOM matrix. However, channel correlation among the users limits the system performance due to destructive interference and ill-conditioned channels. Ill-conditioned channels are not invertible, affecting MUSE's linear receiver which performs Zero-Forcing equalization.

Cyclic Shift Delays (CSD) have been used in wireless communications to decorrelate transmitted signals by introducing diversity. At the time of training, the same preamble signals are transmitted by multiple antennas. Even though a rich multipath environment decorrelates the simultaneously transmitted signals at the receiver, if the same signal is transmitted by multiple antennas, they can experience correlation and result in power fluctuations and undesirable beamforming effects [5].

Table 1: 802.11n Cyclic Shift Delay.

Number of Streams	CSD for TX 1 (ns)	CSD for TX 2 (ns)	CSD for TX 3 (ns)	CSD for TX 4 (ns)
1	0			
2	0	-400		
3	0	-400	-200	
4	0	-400	-200	-600

The 802.11n standard introduces Cyclic Shift Delays to alleviate unintended beamforming. CSDs are applied in the frequency domain as shown in Equation (5), where $S(f)$ is the Fourier transform of $s(t)$ and T_{CSD} is the corresponding CSD value. The Cyclic Shift Delay (T_{CSD}) is applied to both preamble and data.

$$S_{CSD}(f) = S(f)e^{-j2\pi f T_{CSD}} \tag{5}$$

CSD in the 802.11 standard are predefined phase shift delays, shown in Table 1, that are applied to each of the transmitting signals. However, contrary to 802.11n where antennas are co-located, MUSE transmitters are separated by distances that are unpredictable, but nonetheless expected to at least be multiple wavelengths. Consequently, we introduce "Arbitrary Cyclic Shift Delay" (aCSD) a flexible CSD design that leverages transmitter separation without requiring a control channel nor synchronization. Channel correlation decreases as transmitters separate. Thus, unlike fixed usage of CSD in 802.11n, we enable distributed users to arbitrarily select a CSD value. Through this design, we overcome the lack of a control channel among the transmitters and provide flexibility to adapt to the diverse channels of the distributed transmitters. Multiple streams can apply the same phase shift value and still obtain multiplexing gains provided by the high multipath environment because of the different paths and propagation delays between the users. This is only possible because users are distributed in space, leading to increasingly uncorrelated channels as compared to co-located antennas.

2.4 Varying Number of Streams

Downlink multi-user and SU-MIMO have traffic generation and transmit opportunity gained by a single entity, in MUSE, *distributed* clients compete for channel access when they are backlogged. Consequently, MUSE does not fix the number of data streams prior to transmission, i.e., the selected group of clients is not assured to all be backlogged. This contrasts with existing 802.11n systems with pre-configured number of spatial streams and a fixed orthogonal mapping matrix.

MUSE PHY design overcomes this by permitting a variable number of data streams, thereby providing flexibility and robustness to unknown backlogged information of grouped users by a dynamic operation of the DOM matrix. In a multi-user transmission with a missing user, not all rows of the full-rank DOM matrix are used. Additionally, the rows of the DOM matrix are not necessarily used in order because the missing users are unknown. The missing and non-ordered rows of the DOM matrix have no impact on the channel estimation.

In MUSE, the number of users in a group is always equal to the number of receive antennas at the AP. Consequently, if all users have traffic to transmit, MUSE will achieve the full-rank multiplexing capacity. However, when one or more users of the group does not have traffic demand or misses the trigger to join the transmission, the extra AP resources serve as receive diversity to increase robustness. Nonetheless, MUSE uplink transmission will always have at least one client with traffic, the one who gained the transmit opportunity.

To explain the dynamic operation of the DOM matrix, we use an example scenario shown in Figure 3 with an AP with two antennas and multiple associated clients. Here a client wins channel access and gives transmit opportunity to a secondary client, in this case the group size is limited to two because of the number of receive antennas at the AP. Both the AP and the primary client expect a 2x2 uplink multi-user transmission and consequently the client transmits two HTLTF and the AP performs the decoding process for two spatial streams. However, only the primary client has traffic to transmit which results in a 2x1 MIMO transmission since user 2 does not transmit.

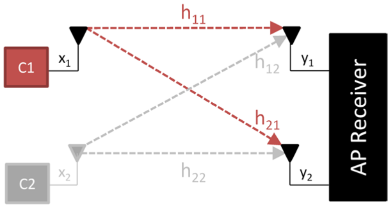


Figure 3: MUSE 2x1 MIMO transmission. AP and C1 expect a 2x2 transmission but C2 has no traffic.

The expected signals from a 2x2 UL MIMO and the AP received signals from this example are:

$$\begin{aligned} y^{sc}_{1,t_1} &= h^{sc}_{11} \cdot HTLTF + \cancel{h^{sc}_{12} \cdot HTLTF} + z^{sc}_{1,t1} \\ y^{sc}_{1,t_2} &= -h^{sc}_{11} \cdot HTLTF + \cancel{h^{sc}_{12} \cdot HTLTF} + z^{sc}_{1,t2} \\ &. \\ y^{sc}_{2,t_1} &= h^{sc}_{21} \cdot HTLTF + \cancel{h^{sc}_{22} \cdot HTLTF} + z^{sc}_{2,t1} \\ y^{sc}_{2,t_2} &= -h^{sc}_{21} \cdot HTLTF + \cancel{h^{sc}_{22} \cdot HTLTF} + z^{sc}_{2,t2} \end{aligned} \tag{6}$$

Observe that the primary client transmitted two HTLTF (t_1 and t_2) expecting a secondary transmitter. The AP processes the received signals and estimates the expected four channels, as follows:

$$\begin{aligned} \hat{h}^{sc}_{11} = \frac{y^{sc}_{1,t_1} - y^{sc}_{1,t_2}}{2 \cdot HTLTF} = \frac{\cancel{2} \cdot \hat{h}^{sc}_{11} \cdot \cancel{HTLTF}}{\cancel{2} \cdot \cancel{HTLTF}} &= \hat{h}^{sc}_{11} \\ \hat{h}^{sc}_{12} = \frac{y^{sc}_{1,t_1} + y^{sc}_{1,t_2}}{2 \cdot HTLTF} &= z^{sc}_{1} \\ \hat{h}^{sc}_{21} = \frac{y^{sc}_{2,t_1} - y^{sc}_{2,t_2}}{2 \cdot HTLTF} = \frac{\cancel{2} \cdot \hat{h}^{sc}_{21} \cdot \cancel{HTLTF}}{\cancel{2} \cdot \cancel{HTLTF}} &= \hat{h}^{sc}_{21} \\ \hat{h}^{sc}_{22} = \frac{y^{sc}_{2,t_1} + y^{sc}_{2,t_2}}{2 \cdot HTLTF} &= z^{sc}_{2}. \end{aligned} \tag{7}$$

Equations (7) indicate that the AP is able to estimate $\hat{h}^{sc}_{11}$ and $\hat{h}^{sc}_{21}$. However, for channel estimates $\hat{h}^{sc}_{12}$ and $\hat{h}^{sc}_{22}$ it just obtains noise. Even with noise estimates for secondary transmitter channels, the AP is able to decode the primary transmitter's data packet and the extra AP antenna resources serve as receive diversity.

MUSE's adaptive usage of preamble-base channel estimation extends to any number of spatial streams. The key is to always permit the maximum available number of data streams supported by the AP. In case not all triggered clients have traffic demand, the extra overhead of preamble symbols is minimal and is not comparable to the overhead of sounding for CSIT feedback. In the previous example, the extra overhead is equal to 1 HTLTF which corresponds to $4\mu s$. The general expression for MUSE extra overhead is

$$(max(N_{ss}) - N_{TX}) \cdot 4\mu s \tag{8}$$

where $max(N_{ss})$ is the max number of data streams, N_{TX} represents the number of transmitting clients where the subtraction of these corresponds to the number of extra HTLTF symbols that each add $4\mu s$ overhead.

3 Medium Access with Arbitrary Group Members

In this section we present a distributed random access and user-grouping protocol for multi-user uplink medium access. We base the design on the capabilities of the MUSE physical layer and target constant overhead that does not increase with the number of users simultaneously served, enabling MUSE scaling to large array sizes of distributed users.

3.1 Association ID Grouping

The MUSE-PHY properties include CSIR-based open-loop multi-user uplink transmission via an arbitrary set of users, provided that the rank of the AP is not exceeded (i.e., the total number of antennas of all clients must not exceed the number of antennas on the AP). Our technique proceeds in two steps.

In the first step, all backlogged users contend for the medium through standardized backoff countdown process. The first user to count to 0, which we refer to as the contention-winning user, wins the medium. The contention-winning user then sends a triggering message to all users with its Association-ID. The triggering message grants a transmit opportunity to a predefined random set of users and serves as the time-synchronization trigger for the multi-user transmission. If a user is out-of-range to the contention-winning user it is deaf to the trigger and misses the opportunistic medium access. Nevertheless, if the deaf-to-trigger user has traffic to transmit it will obtain a transmit opportunity when its backoff counter expires. As described in Section 2, MUSE-PHY is robust to deaf-to-trigger users.

In the second step, we join an arbitrary set of users to the first user as follows. According to the IEEE 802.11 standard, upon association to an AP, users are assigned an Association ID. MUSE leverages this association ID for user selection and grouping by considering this to be an arbitrary index for each user. For MUSE grouping, the AP informs the network the total number of associated users, i.e., the Max-Association-ID. We join as many users as possible to the contention-winning user as limited by the AP rank. For example, if the receiving AP has N antennas, the medium contention-winning user triggers $N-1$ additional users with the successive Association-ID. If the contention-winning user ID is towards the end such that there are not $N-1$ successive IDs available, we consider IDs to be circular and wrap back to ID 1 as illustrated in Figure 4. Realization of circular ID grouping is possible because the AP informs all users of the Max-Association-ID.

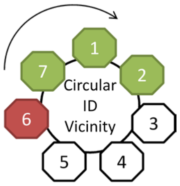


Figure 4: MUSE's circular association ID.

After receiving the data from the multiple users, the AP acknowledges the successfully decoded packets *independently* such that each user can determine its supported and desired type of acknowledgment, such as Block ACK immediate or delayed.

To illustrate, consider the example in Figures 4 and 5, in which the AP has 4 antennas and there are a total of 7 users in the network of which four can transmit at the same time. As shown in the timeline of Figure 5, the user with ID 6 wins the medium access contention with the smallest backoff counter of 3. After the backoff expiration, the contention-winning user triggers three additional user transmissions in order to reach maximum rank of 4. However, because there are a total of seven users, ID 6 will be grouped with users of IDs: 7, 1 and 2, as shown in related Figure 4.

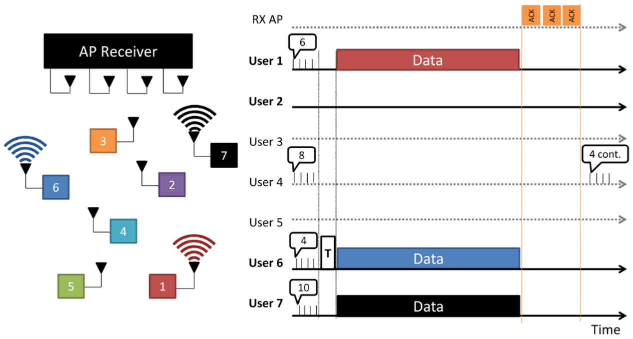


Figure 5: MUSE 4x3 MAC example, with a 4-antenna AP and seven users.

The triggering transmission acts as a beacon packet which informs all users in the network the medium winning-user ID, as shown in the timeline of Figure 5. With the ID of the contention-winning user, all users know if their ID falls within the $N-1$ vicinity IDs to obtain an opportunity for contention-free medium access by joining the multi-user transmission. If stations having the vicinity IDs have traffic available, they transmit immediately after receiving the trigger beacon and are synchronized via the timing of the beacon.

Because we use Wi-Fi contention to select the originating member of the group, we inherit the fairness properties of Wi-Fi. Further, MUSE resets backoff counters for all users that accessed the medium even if these were granted medium access without expired backoff counters. As seen in the Figure 5, users 7 and 1 had backoff counters of 10 and 6 respectively, when group access was granted. In this case, since medium access was obtained, a new backoff counter must be chosen for new packets. Likewise, because each user has the same probability to win the contention, each user will be grouped the same number of times on average, provided that all users are fully backlogged.

3.2 Inter-User Stream Coordination

MUSE-PHY requires that the selected users start transmission at the same time. Such time synchronization is achieved by the triggering beacon, where the trigger message serves as a Clear-To-Send to the users in a group to start transmission SIFS time after its reception.

To coordinate the users for MUSE-PHY to enable uncorrelated channels, each transmitting user must know which part of the Dynamic Orthogonal Mapping matrix to apply. Consequently, we utilize the association IDs to classify the transmitting users such that their IDs further serve as the "Stream Number" assignment. In particular MUSE-PHY requires a user to map to a Stream Number which determines the Dynamic Orthogonal Mapping matrix to apply. In the example of Figure 5, contention-winning user 6 is established as Stream 1, and consecutive users in the group 7, 1 and 2 are assigned Stream 2, 3 and 4 respectively. With an assigned Stream number, each user will apply the corresponding Dynamic Orthogonal Mapping matrix. In the example, only users 6, 7 and 1 have traffic to transmit these are Stream No. 1-3. These streams apply row 1-3 of Dynamic Orthogonal Mapping matrix matrix shown in Equation (3) and choose their best suited Arbitrary Cyclic Shift Delay for their location from values -100 ns to -700 ns.

3.3 Group Adaptation and Backlog

With MUSE-MAC, the triggering user is guaranteed to be backlogged as only backlogged users initiate medium contention. However, it is possible that one or more of the remaining arbitrarily selected users are not backlogged. MUSE-PHY ensures that the AP can decode the received transmission for any subset of the random group members being backlogged, from 1 to all. Nonetheless, if traffic is sparse, fixed group selection without incorporating backlog state will result in a throughput penalty as non-full-rank uplink transmissions will occur. However, if traffic is fully backlogged, all MUSE uplink transmissions will be full rank.

For sparse traffic, groups can be updated through reassignment of association IDs. Today, association IDs are reassigned to users via the procedure of Reassociation Request and Response. A Reassociation Request frame is sent by a station (STA) to an AP when the STA already associated to the Extended Service Set (ESS) has left the cell for a short duration and wants to rejoin or when a STA wants to associate to another AP in the same ESS [13]. The AP responds to this request with an Association Response frame which assigns a new Association ID to the STA.

To update group assignments the AP can prompt the Reassociation Request and Response procedure. The AP being the receiver has knowledge of which users to group to increase the probability of full-rank uplink transmissions as these have previously transmitted uplink traffic. A traffic-based group can be formed by the AP by prompting the Reassociation Request and Response procedure to the selected users and assign these continuous Association IDs. Optimized user selection and grouping is a large area of study in MU-MIMO [14, 15, 16, 17] and such techniques could be extended to MUSE.

4 Implementation and Evaluation

In this section we present the implementation and experimental evaluation of MUSE for an indoor WLAN scenario. Our evaluation focuses on MUSE scalability, user orthogonality and MAC user grouping and performance.

4.1 MUSE Implementation

MUSE Implementation on WARP. We implement MUSE on a software defined radio platform that enables Over The Air (OTA) experiments [18]. The platform, Wireless Open-Access Research Platform (WARP), supports a programming environment that performs OTA data transmission and reception and offline processing.

We implement a complete 802.11n OFDM physical layer with modulation rates of 6 Mbps (BPSK), 12 Mbps (QPSK), and 24 Mbps (16-QAM) and include mechanisms for heterogeneous modulation rates among streams. [3] This feature enables each uplink transmitter to select its highest possible bit rate, without requiring that all users make the same selection. For ease of implementation we consider only half rate modulations. We implement the complete suite of MUSE's PHY techniques Dynamic Orthogonal Mapping matrix and variable Arbitrary Cyclic Shift Delay on our platform. Our implementation permits 1 to 4 concurrent spatial streams transmissions enabling full rank MIMO transmissions from 1x1 to 4x4 and alternative modes e.g., 2x4 and 3x4.

Testbed Setup. We use the 2.484 GHz radio channel, i.e., channel 14 (currently unused spectrum) for all experiments. All experiments are conducted at night to ensure experimental repeatability with minimal unaccounted for factors enabling us to isolate the effects of inter-stream and inter-user interference. We perform experiments with multiple node topologies with over 20 client locations in a conference room setting depicted in Figure 6. For each experiment, the topology setup is specified at each evaluation. The setup is configured from a 1x1 to a 4x4 MU-MIMO system depending on the evaluation being performed. Each MUSE client node is deployed with a single WARP board running independent RF clocks. Phase unsynchronized clients in our testbed demonstrate the nonessential need of phase synchronization among the distributed transmitters. However to emulate the beacon triggering system of MUSE, we time-synchronize the transmission of the distributed users through triggering cables to all client-nodes that activate all users' transmissions at the same time.

[3] Our system performance is not limited to a maximum of 24 Mbps. This is just an evaluation platform constraint.

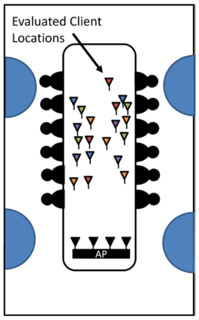


Figure 6: Experimental conference room setting with evaluated locations.

4.2 Multiplexing Gains and Scalability

MUSE targets a linear scaling increase that matches the number of simultaneous uplink transmissions to the number of receiver antennas at the AP. Multiplexing gains and scaling are limited by inter-stream interference and channel-correlation between the users. In this section we evaluate the ability of MUSE's Dynamic Orthogonal Mapping matrix and Arbitrary Cyclic Shift Delay to achieve multiplexing gains and permit linear scaling as the number of transmitters grows.

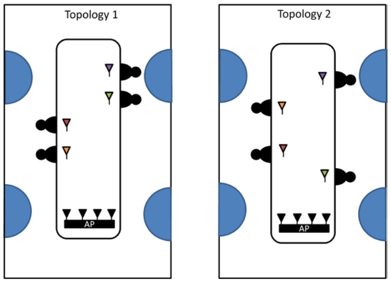


Figure 7: Experimental Setup for Scalability Evaluation.

For this experiment, the system setup consists of 4 independent transmitting users and an AP receiver with 4 antennas, as depicted in Figure 7. The four transmitters are placed at two topologies shown in Figure 7, where at each topology 2x2, 3x3 and 4x4 transmissions are performed with the AP. The topologies are chosen to represent a conference setting with users sitting next to each other in topology 1 and spread out by one or more chairs in setting 2. A total of 2000 packets are transmitted at 24 Mbps (16-QAM) per setting (1x1, 2x2, 3x3, 4x4), where the number of active concurrent transmissions increases from 1 to 4.

Figure 8 shows the PHY multiplexing gains achieved by MUSE in the evaluated scenarios. The y-axis depicts the throughput gains in percentage, where the aggregate throughput of the transmitting users is normalized by the evaluated channel PHY rate (24 Mbps). The aggregate throughput is obtained from the Packet-Error-Rate of each stream. The x-axis depicts the number of active concurrent transmitters.

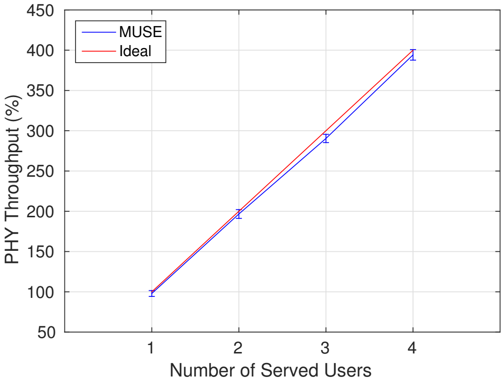


Figure 8: MUSE scalability shown by PHY throughput as number of users increase.

Figure 8 indicates that MUSE achieves a data rate that linearly increases with the number of users for the evaluated rates and scenario. In particular, the Dynamic Orthogonal Mapping matrix and Arbitrary Cyclic Shift Delay successfully isolate the transmitting streams for decoding at all tested locations. Specifically, MUSE achieves on average 197%, 290% and 395% aggregate PHY throughput for 2 to 4 concurrent served users respectively. The error bars indicate that in some of the tested locations, full-rank rates were achieved, whereas the lowest rates fall within 10% of ideal full-rank PHY performance.

The scalability shown for 24 Mbps rates in Figure 8 holds for various SNR values and data rates. We demonstrate the scalability of MUSE's capacity for varying SNR values in Figure 9. To calculate MUSE capacity we use the generalized Shannon capacity formula for M transmit antennas and N received antennas given by $C(bps/Hz) = \log 2[\det(I_N + (\rho/M)(HH^*))]$ [19], where H^* is the conjugate transpose of H, I_N is the NxN identity matrix and ρ is the average SNR. We use the channels (H) from the OTA experiments described for Figure 8 where H is measured at the receiver after applying MUSE-PHY techniques. We calculate the capacity for each subcarrier at each transmission for SNR values from 0 to 20 and depict the average capacity per MIMO setting in Figure 9.

MUSE-PHY enables large multiplexing gains, however perfect linear scaling is not realized in all settings due to some residual channel correlation. Nevertheless, significant gains are achieved, within 4 to 6% of ideal for 2x2, 9 to 18% of ideal for 3x3 and 13 to 27% of ideal for 4x4. We observe for an SNR value of 20 dB, MUSE capacity increases from 9.6 bps/Hz for a 1x1 to

18.45 bps/Hz for a 2x2, 26.11 bps/Hz for a 3x3 and 33.48 for a 4x4. The theoretical maximum rates for the evaluated channels range from 126 Mbps to 410 Mbps for an SNR of 10 dB and 193 Mbps to 669 Mbps for an SNR of 20 dB.

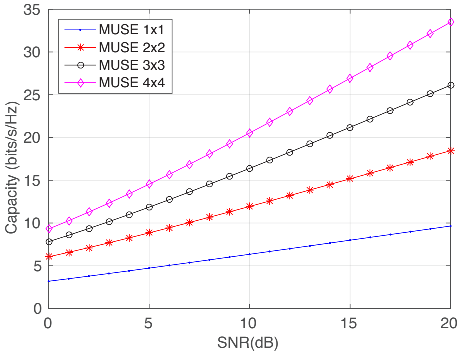


Figure 9: MUSE capacity for OTA measured channels.

These multiplexing gains demonstrate the scalability of MUSE and the ability of MUSE-PHY to enable simultaneous distributed transmitters in high multipath scenarios with well-conditioned channels. MUSE-PHY techniques can achieve full-rank multiplexing gains without need for CSIT, while not being affected by interference or correlated channels.

4.3 Signal Decorrelation

A key to achieve multiplexing gains is decorrelating users signals via the Arbitrary Cyclic Shift Delay (aCSD) for correct channel estimation. In contrast to co-located use of cyclic shifts delays, MUSE distributed transmitters have diverse wireless-environment from sparse locations which lead to different signal-paths to the AP. Here, we evaluate the effect the different aCSD settings have on performance and signal correlation of distributed transmitters with distinct locations. Through this evaluation we derive the effect aCSD has on performance when transmitters are distributed in space and determine the relation aCSD has with user locations in relation to other users and the AP.

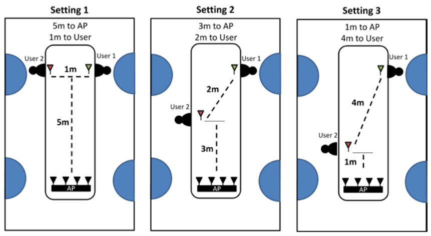


Figure 10: Setup for aCSD Evaluation.

In this experiment we fix the number of users to two, and perform 2x2 uplink multi-user transmissions at 16-QAM (24 Mbps) each. As shown in Figure 10, we evaluate 3 settings where we fix user 1 at a 5 m distance to the AP and vary User 2 distance to the AP from 5 m (equal distance to AP as user 1) to 3 m and 1 m. The distance between users is varied as User 2 is moved closer to the AP as depicted in Figure 10. At each location all 17 aCSD values are evaluated ranging from 0 ns to -800 ns in 50 ns steps. Only 17 values are possible since the IEEE 802.11 standard's Cyclic Prefix size is defined as 16 symbols resulting in a max of 800 ns aCSD.

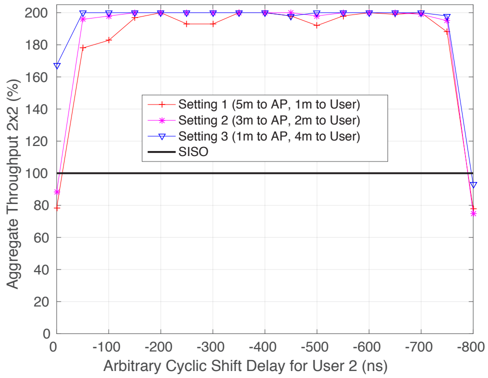


Figure 11: Effect of aCSD values on performance and channel correlation of distributed transmitters on three evaluated settings.

Figure 11 depicts the effect varying aCSD values depicted on the x-axis have on the system performance shown in the y-axis. System performance is represented as a percentage where the aggregate system throughput is normalized by the evaluated SISO rate of 24 Mbps.

The figure indicates that MUSE achieves the desired performance of 200% PHY utilization across most aCSD values from -100 ns to -700 ns. However, signal correlation can be observed in Setting 1 in Figure 11 where performance drops exists for some aCSD values. From the 3 evaluated scenarios, we observe that user channels are more correlated at the scenario of Setting 1 where both users are at 5 m distance to the AP. As a result, we observe that sparse user placement of settings 2 and 3 benefits performance by providing uncorrelated channels, allowing usage of any aCSD value between 100 ns to 700 ns.

In the presence of correlated channels as seen at close user proximity, specific CSD values are required, as used today in 802.11n system where transmitters are co-located. We observe a value of -400 ns is suitable at any of the 3 evaluated settings, since -400 ns delay corresponds to a 90 degree phase shift which presents the best results when users have highly correlated channels.

Lastly, we observe that aCSD values of 0 ns or 800 ns which both correspond to applying no phase-shift, obtain very low PHY throughput values that fall below SISO rates. Low PHY throughput values for 0 ns or 800 ns aCSD values are caused by correlated training signals leading to erroneous channel estimation at the receiver. However, we observe that the throughput values for these aCSD values are not 0. This indicates that the high multipath environment provides independent channels. However in the case where the training signals are transmitted through multiple antennas, these may result in correlated signals leading to beamforming effects creating nulls or signal maximum at receive antennas. We can conclude that when aCSD is applied to the distributed transmitters, we effectively induce multipath in the form of time delay of the simultaneous signals which leads to signal decorrelation. The spatial sparsity of the distributed transmitters allows any aCSD value to provide the required multipath to isolate the streams for high multiplexing gains.

4.4 User Grouping and Medium Access

MUSE MAC performs user grouping without knowledge of channel estimates or SINR among the selected users. MUSE grouping has minimal overhead by leveraging Association ID to enable opportunistic medium access to users with neighboring association IDs. Here, we evaluate random user grouping used by MUSE as compared to perfect user selection, identified experimentally via exhaustive search.

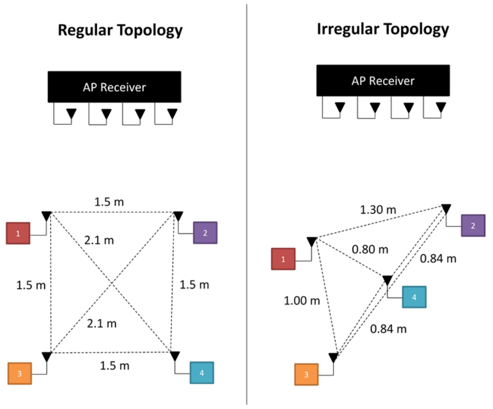


Figure 12: Setup for Random User Grouping Evaluation.

We evaluate user grouping by performing 2x2 UL MU-MIMO transmissions where each user transmits at the max rate of our evaluation platform of 24 Mbps (16-QAM). We perform an exhaustive evaluation of the grouping combination for a pair of users in a total set of 4 users distributed in space. By evaluating all possible user grouping combinations, i.e., 1-2, 1-3, 1-4, 2-3, 2-4, 3-4, we can analyze the difference in performance from the distinct groups. We evaluate all grouping combinations in the two topologies shown in Figure 12 where 500 packet transmissions are evaluated per group combination. We evaluate a regular topology shown on the left of Figure 12 that emulates four users in a conference room sitting at equal distances. Additionally, we evaluate an irregular topology shown on the right of Figure 12 where distances between the users vary from 80 cm to 1.3 m.

Table 2: Aggregate PHY Throughput for varying user grouping.

Grouping	Regular Topology		Irregular Topology	
	Dist. (m)	Aggregate Capacity (%)	Dist. (m)	Aggregate Capacity (%)
1-2	1.50	**200.0**	1.30	**199.6**
1-3	1.50	**191.4**	1.00	**199.3**
1-4	2.10	**199.6**	0.80	158.7
2-3	2.10	146.0	1.68	137.7
2-4	1.50	132.0	0.84	128.7
3-4	1.50	**194.66**	0.84	**199.3**

Table 2 shows the aggregate throughput results per grouping combination for evaluated typologies shown in Figure 12. The aggregate performance results are shown in percentage which represent the aggregate throughput normalized by the single-stream PHY rate of 24 Mbps, where a value of 200% represents that 2x2 full multiplexing gains (48 Mbps in our evaluation platform) are achieved. The throughput results are obtained through packet-error-rate from all transmitted packets.

The results indicate that in the evaluated regular topology, 4 of the 6 groupings (marked in bold) obtain 91 - 100% performance increase over MISO transmissions and only 2 grouping combinations achieve 32-46% increase. In the evaluated irregular-topology we observe that 3 out of 6 grouping combinations (marked in bold) are within 1% of ideal 2x2 throughput. However, 3 out of 6 grouping combinations of the evaluated irregular topology obtain 28-58% percentage increase over MISO performance. In groupings with lower gains, only one stream shows higher packet-error-rate resulting in undecodable packet. However, because each user transmits independent data (no beamforming) the loss of one stream does not affect the performance of the other and thus we observe aggregate throughput values of 132 to 146% and not below 100% (MISO performance). Consequently, to reduce packet-error-rate and increase robustness and performance of ill-conditioned channels a lower modulation is required. Existing multi-user rate adaptation solutions such as TurboRate [20] can be adapted by MUSE to increase system performance in the case a user experiences an ill-conditioned channel.

Additionally, we observe from the results of Table 2 that varying distances between users does not affect performance. In both evaluated topologies we observe that low gains are achieved by short inter-user-distance of 0.8 m and larger inter-user-distance of 2.10 m. This observation demonstrates that in the evaluated scenario, sub-optimal grouping is not related to the distance between users but instead is dependent on the user's channel conditions which varies according to room placement. We conclude that multipath of the evaluated indoor scenario permits full rank multiplexing gains at most evaluated locations without need of CSI or SINR knowledge. Ideally, a MUSE system alternates among groupings with well-conditioned channels such as grouping 1-2, 1-3 and 3-4 in the evaluated scenario. However, there exist locations that may encountered ill-conditioned channels which lead to lower multiplexing gains. Nonetheless MUSE can be made robust to bad grouping by smartly adapting modulation rate for a selected bad user or reassignment of Association IDs as explained in Section 3.

4.5 MAC Overhead and Performance

Here, we evaluate the scalability of the net system throughput incorporating both physical layer throughput and MAC overhead. As a baseline, we also consider today's Wi-Fi, single-user IEEE 802.11n, which in this scenario uses a single antenna transmitter and multi-antenna receiver (SIMO). In this case, sequential SIMO transmissions are performed and the antennas at the AP are used for receive diversity. Moreover, we compare to uplink multi-user systems SAM [11] and Signpost [12].

We first evaluate MAC-layer overhead of MUSE, 802.11n SIMO, SAM and Signpost. The MAC overhead for each evaluated system is given by

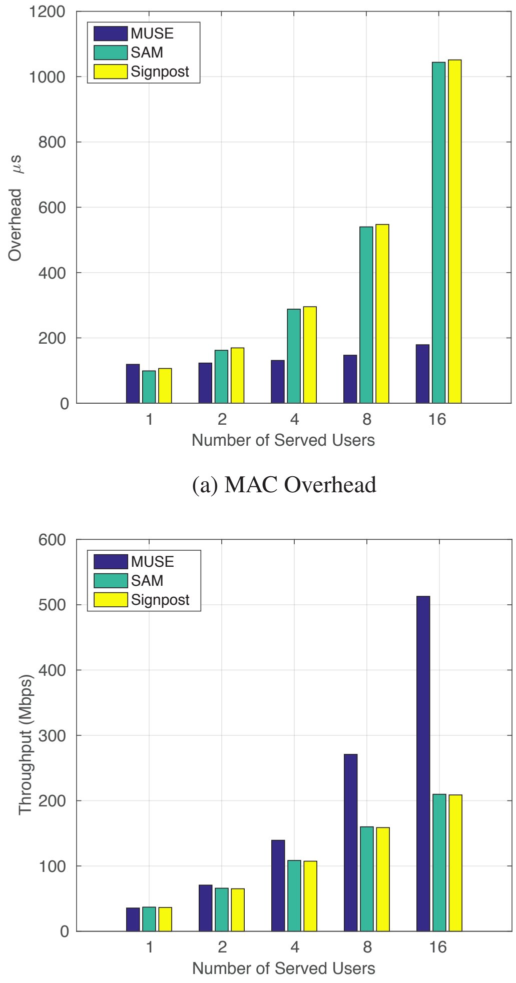
$$MUSE = BO_t + ACK_t + P_t + Trig_t \tag{9}$$

$$SIMO = N*(BO_t + ACK_t + P_t) \tag{10}$$

$$SAM = (N*BO_t) + ACK_t + P_t \tag{11}$$

$$SIGNPOST = (N*BO_t) + ACK_t + P_t + S_{NDPA} \tag{12}$$

where N is the number of served clients, BO_t is the mean backoff time of 7 slots (minimum contention, 15 slots, window divided by 2), ACK_t is the ack time and P_t is the signal preamble time. In the expression for MUSE's overhead given by Equation (9), $Trig_t$ represents the overhead time for the triggering packet sent by the contention-winning user to enable the multi-user transmission. Further, all baseline systems employ sequential contention for each group member of the multi-user transmission, this is shown by $(N*BO_t)$ in Equations (11) and (12). Additionally, Signpost includes a sounding packet (NDPA) of 7.4 μs for signpost calculation.



(a) MAC Overhead

(b) MAC Throughput

Figure 13: Medium Access Overhead and Throughput Scaling.

Figure 13a depicts the MAC overhead vs. the number of users for MUSE and the baseline systems. (SIMO is not depicted as SIMO's overhead rapidly increases as the numbers of users increase as shown in Equation (10) and alters Figure 13a's scale.) We observe that MUSE's overhead remains close to constant as the number of users increases because contention is only performed once per multiuser transmission as shown in Equation (9). Overhead slightly increases (from 119 μs to 179 μs) with rank because the number of training symbols for channel estimation increases with the number of users. The baseline systems' sequential contention $(N*BO_t)$ increases linearly with N, exponential for the depicted AP antenna numbers. Figure 13a shows baseline overhead increased 504 μs from 8 to 16 users, meanwhile MUSE overhead remains close to constant increasing only 32 μs from 8 to 16 users. Thus, even though MUSE overhead is greater for a small number of users, it remains close to constant leading to the best scaling with user population.

Second, we compare MAC throughput of MUSE with IEEE 802.11n SIMO, SAM and Signpost. All systems are simulated with 100 OFDM symbol packets with 6 bits per symbol for a 20 MHz channel, resulting in data rate of 54 Mbps with no transmit collisions (0 Packet-Error-rate). Figure 13b depicts MUSE throughput linear increase, e.g. MUSE throughput increases from 140 Mbps for 4 concurrent users to 513 Mbps for 16 users. However, throughput of baseline systems does not increase linearly as MUSE, e.g., for 8 users baseline throughput is 158.8 Mbps and only increases by 50 Mbps (1/3) when doubling the number of users. Throughput scalability of baseline systems is limited by increasing overhead (shown in Figure 13a). Finally, MUSE's marginal gains over baseline increases with rank: with a low number of users (1-2), MUSE performance is equal or slightly worse than baseline. However, as the number of users increases, MUSE's gains over the baseline escalate where we observe gains of 2.5x for 16 antenna APs compared to the multi-user baseline. This evaluation demonstrates the importance of constant overhead in scaling multi-user uplink medium access.

5 Prior Work

MUSE is the first PHY and MAC system that enables scalable full-rank uplink multi-user multiplexing without requiring a control channel. Here, we contrast MUSE to prior work in both downlink and uplink multi-user transmission.

Multi-User Downlink. There is a vast body of theoretical [3] and experimental [1, 2, 4, 5] research that demonstrates the multiplexing gains available in downlink multi-user MIMO. Moreover, downlink multi-user WLAN transmission was standardized in 2014 in IEEE 802.11ac [7]. In both research and commercial systems, Transmit beamforming (TxBF) is used to uncorrelate users' channels and serve multiple users simultaneously. TxBF uses CSIT to form weight vectors that isolate the data streams of the different users.

In contrast to downlink multi-user transmission where data is sourced from a single AP, uplink multi-user data is sent from multiple devices that are independent and spatially distributed. Distributed transmitters have independent clocks that cause channel estimation to become stale or obsolete as their phases drift differently and independently. In TxBF all the space-time streams are combined and multiplied by a matrix of steering vectors to produce the input to the transmit chains. However, distributed transmitters have no control channel among them which prohibits the stream combination in TxBF. Further, CSI overhead increases linearly with rank which fails to achieve MUSE's scaling goal for uplink multi-user transmission.

Multi-User Uplink. There is limited prior work on distributed multi-user uplink WLANs and to date it is neither standardized nor commercialized for WLANs. Existing WLAN uplink multi-user solutions [12, 11, 21] enable multiplexing through sequential contention for each group member. However, sequential contention incurs control overhead (training and contention) that increases linearly with group size. In contrast, MUSE-MAC scalable grouping eliminates the need for a group-selection control procedure via a single contention (fixed-overhead).

In aforementioned solutions stream isolation is achieved in three different ways: by CSI based prealigned orthogonal directions in [12], successive interference cancellation and staggered preambles in [11] and interference alignment and cancellation [22] in [21]. In contrast, MUSE-PHY enables full-rank multiplexing with standard compliant channel estimation from all distributed users, with temporally overlapped preambles for an arbitrary number of users.

6 Conclusions

In this paper we introduce MUSE, the first system to achieve full-rank uplink multi-user capacity in a fully distributed and scalable manner without a control channel. In MUSE, no control messages are used for channel estimation, CSI feedback and channel-based user selection. We design MUSE-PHY which decorrelates users' signals through Arbitrary Cyclic Shift Delays, enables preamble-based clean channel estimation at the receiver with the Dynamic Orthogonal Mapping matrix and adapts to variable traffic demand of distributed transmitters. We design a fixed-overhead scalable MUSE-MAC that enables a multi-user multi-stream transmission through a single medium access contention. MUSE-MAC attaches a random set of additional users to the winning-user and assures the rank of the group equals the number of antennas at the AP. Our experiments demonstrate full-rank multiplexing gains in the evaluated scenarios that show linear gains as the number of users increase. Our experimental results show an average PHY capacity utilization of 197%, 290% and 395% for 2 to 4 concurrent users respectively with evaluated rates and maintain constant overhead as the number of users increases.

7 Acknowledgments

This research was supported by Cisco Systems, Intel, the Keck Foundation, and by NSF grants CNS-1514285, CNS-1444056, CNS-1126478 and CNS-1012831.

References

[1] C. Shepard, H. Yu, N. Anand, E. Li, T. Marzetta, R. Yang, and L. Zhong, "Argos: Practical many-antenna base stations," in *Proc. ACM MobiCom*, 2012.

[2] X. Xie, E. Chai, X. Zhang, K. Sundaresan, A. Khojastepour, and S. Rangarajan, "Hekaton: Efficient and Practical Large-Scale MIMO," in *Proc. ACM MobiCom*, 2015.

[3] A. Goldsmith, S. A. Jafar, N. Jindal, and S. Vishwanath, "Capacity limits of MIMO channels," *IEEE Journal on Selected Areas in Communications*, 2003.

[4] E. Aryafar, N. Anand, T. Salonidis, and E. W. Knightly, "Design and experimental evaluation of multi-user beamforming in wireless LANs," in *Proc. ACM MobiCom*, 2010.

[5] E. Perahia and R. Stacey, *Next Generation Wireless LANs: 802.11n and 802.11ac*. Cambridge University Press, 2013.

[6] O. Bejarano, E. W. Knightly, and M. Park, "IEEE 802.11ac: from channelization to multi-user MIMO." *IEEE Communications Magazine*, 2013.

[7] IEEE 802.11ac, "IEEE Standard for Information technology Local and metropolitan area networks Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput," *IEEE Std 802.11ac-2014*.

[8] "Qualcomm VIVE with Qualcomm MU EFX Multi-User MIMO." [Online]. Available: http://www.qca.qualcomm.com/products/qualcomm-vive/

[9] "Quantenna communications." [Online]. Available: http://www.quantenna.com/

[10] B. Crow, I. Widjaja, J. G. Kim, and P. Sakai, "IEEE 802.11 Wireless Local Area Networks," *IEEE Communications Magazine*, Sep 1997.

[11] K. Tan, H. Liu, J. Fang, W. Wang, J. Zhang, M. Chen, and G. M. Voelker, "SAM: enabling practical spatial multiple access in wireless LAN," in *Proc. ACM MobiCom*, 2009.

[12] A. Zhou, T. Wei, X. Zhang, M. Liu, and Z. Li, "Signpost: Scalable MU-MIMO signaling with zero CSI feedback," in *Proc. ACM MobiHoc*, 2015.

[13] IEEE 802.11-2012, "IEEE Standard for Information technology Local and metropolitan area networks Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," *IEEE Std 802.11-2012*.

[14] M. Esslaoui, F. Riera-Palou, and G. Femenias, "A fair MU-MIMO scheme for IEEE 802.11ac," in *International Symposium on Wireless Communication Systems*, 2012.

[15] Z. Shen, R. Chen, J. Andrews, R. Heath, and B. Evans, "Low complexity user selection algorithms for multiuser MIMO systems with block diagonalization," *IEEE Transactions on Signal Processing*, 2006.

[16] N. Anand, J. Lee, S.-J. Lee, and E. W. Knightly, "Mode and User Selection for Multi-User MIMO WLANs without CSI," in *Proc. of IEEE INFOCOM*, 2014.

[17] T. Tandai, H. Mori, and M. Takagi, "Cross-layer-optimized user grouping strategy in downlink multiuser MIMO systems," in *Proc. IEEE VTC*. IEEE, 2009.

[18] "WARP project." [Online]. Available: http://warpproject.org

[19] G. J. Foschini, "Layered space-time architecture for wireless communication in a fading environment when using multi-element antennas," *Bell labs technical journal*, 1996.

[20] W.-L. Shen, K. C.-J. Lin, S. Gollakota, and M.-S. Chen, "Rate adaptation for 802.11 multiuser MIMO networks," *IEEE Transactions on Mobile Computing*, 2014.

[21] K. C.-J. Lin, S. Gollakota, and D. Katabi, "Random access heterogeneous MIMO networks," *Proc. ACM SIGCOMM*, 2011.

[22] S. Gollakota, S. D. Perli, and D. Katabi, "Interference Alignment and Cancellation," *Proc. ACM SIGCOMM*, 2009.

BeamSpy: Enabling Robust 60 GHz Links Under Blockage

Sanjib Sur, Xinyu Zhang, Parmesh Ramanathan
University of Wisconsin-Madison

Ranveer Chandra
Microsoft Research

Abstract

Due to high directionality and small wavelengths, 60 GHz links are highly vulnerable to human blockage. To overcome blockage, 60 GHz radios can use a phased-array antenna to search for and switch to unblocked beam directions. However, these techniques are reactive, and only trigger after the blockage has occurred, and hence, they take time to recover the link. In this paper, we propose *BeamSpy*, that can instantaneously predict the quality of 60 GHz beams, even under blockage, without the costly beam searching. *BeamSpy* captures unique spatial and blockage-invariant correlation among beams through a novel prediction model, exploiting which we can immediately select the best alternative beam direction whenever the current beam's quality degrades. We apply *BeamSpy* to a run-time *fast beam adaptation* protocol, and a *blockage-risk assessment* scheme that can guide blockage-resilient link deployment. Our experiments on a reconfigurable 60 GHz platform demonstrate the effectiveness of *BeamSpy*'s prediction framework, and its usefulness in enabling robust 60 GHz links.

1. Introduction

The 60 GHz millimeter-wave (mmWave) band, with up to 7 GHz of unlicensed spectrum, offers the foundation for a new wave of applications, such as uncompressed video streaming, instant file sync, wireless datacenters and wireless fiber-to-home access. Demand for such data-hungry applications, together with worldwide availability of the 60 GHz band, have fueled multiple standardizations, *e.g.*, IEEE 802.11ad [1], 802.15.3c [2] and ECMA-387 [3]. A similar paradigm has been advocated by industry to realize next-generation multi-Gbps cellular networks [4].

To counteract strong signal attenuation, a pair of 60 GHz transceivers can establish a *link* by forming highly directional *beams* using phased-array antennas. However, the pseudo-optical nature of a beam renders it extremely sensitive to blockage, especially in indoor deployments with heavy human activity [5]. Existing 60 GHz network standards have built in a counter-measure — a phased-array antenna can electronically switch between a prescribed set of beam directions, and bounce the signals off opportunistic reflectors, and thus detouring the blockage. But two new challenges emerge. (1) *Run-time overhead.* Searching for alternative Tx/Rx beam directions involves a tedious scanning and signaling procedure. The overhead grows almost quadratically with the number of beam directions. Since a 60 GHz phased array typically generates tens to hundreds of beams, the overhead can easily overwhelm the precious channel time of a multi-Gbps link [6, 7]. (2) *Lack of outage prevention.* Beam searching/switching can only react after blockage occurs, which may have already caused detrimental effect on application and transport layer protocols.

In this paper, we propose *BeamSpy* to meet the above challenges. *BeamSpy* can predict the quality of alternative beams by only inspecting the channel response of the current beam used by the receiver. *BeamSpy*'s prediction mechanism leverages two fundamental properties of 60 GHz links: (1) *Channel sparsity:* no matter how many beam directions are available, the transmitter can only reach the receiver via a small set of dominating signal *paths* [7–10]. (2) *Spatial correlation:* the channels formed by different pairs of beams are often correlated, and the correlation remain unaffected by blockage, according to our measurement study (Sec. 3).

BeamSpy's prediction framework is model-driven. It exploits *channel sparsity* and models the channel between the transmitter and receiver using only a discrete set of signal paths, a set that we refer to as *path skeleton*. Furthermore, *BeamSpy* characterizes the *spatial correlation* by modeling the way different beam directions share the same *path skeleton*. The model takes into account the joint effect of phased-array beamforming and 60 GHz channel distortions. The modeling parameters are extracted from a one-time measurement, and invariant under blockage. Using *BeamSpy*, the Tx/Rx radios can instantaneously predict the best alternative beam direction whenever the current beams' quality drops.

BeamSpy can become a core engine for a wide range of 60 GHz protocols involving beam searching. In this paper, we apply *BeamSpy* to design two such protocols that facilitate reliable 60 GHz networking in blockage-prone indoor environment. (1) *Fast beam adaptation:* A link recovery protocol that can approximate the 802.11ad beam searching with a single implicit probing, thus evading the run-time overhead. (2) *Link outage prediction.* A risk-assessment algorithm that predicts the likelihood of link outage under blockage, thus offering guidelines for deploying 60 GHz links in a blockage-proof way.

We validate *BeamSpy* on a custom-built 60 GHz radio platform [7], along with trace-driven emulator running unmodified TCP/IP and application stack. Our experimental results demonstrate that: (1) *BeamSpy*'s prediction framework can effectively forecast quality of all beams by inspecting only one, with an average RSS prediction error of 0.02 to 1.2 dB. (2) *BeamSpy*'s fast beam adaptation protocol can efficiently identify the best beam pair, if any, that can overcome human blockage. It achieves comparable throughput performance with an oracle that knows exact beam quality, and outperforms 802.11ad significantly in application tests. (3) At deployment time, *BeamSpy*'s outage risk analysis can effectively assess vulnerability to human blockage and recommend a re-deployment when necessary.

General properties of 60 GHz channels, and the possibility of overcoming blockage via beam switching, are already well known [7, 11–13]. The key contribution of *BeamSpy* lies in a measurement-based and model-driven framework to help combat blockage without any extensive beam searching. In summary, our contribution breaks down into the following three aspects,

(1) **A measurement study to understand the unique properties of 60 GHz channel that are pertinent to the predictability of 60 GHz beamforming performance** (*Sec. 3*). To the best of our knowledge, we are the first to perform a principled study of the *blockage-invariant spatial correlation* between 60 GHz beams, which roots in an interplay between extremely sparse 60 GHz spatial channels and phased-array beamforming.

(2) **A novel prediction framework that can capture the spatial correlation between beams using a *path skeleton* model, and can predict the performance of different beams without explicit probing** (Sec. *4*). The framework is validated through extensive experiments on a 60 GHz testbed.

(3) **Applications of the prediction framework to design robust 60 GHz indoor networks** (*Sec. 5*). We design the first risk assessment protocol to predict the vulnerability of blockage at deployment time, and a fast beam adaptation protocol to efficiently overcome blockage at run-time.

2. Background & Motivation

2.1 60 GHz Channel and Impacts of Blockage

Due to smaller wavelength, a 60 GHz mmWave link suffers from 28 dB of higher propagation loss than a 2.4 GHz WiFi link. This disadvantage is compensated by using high-gain phased-array antenna, with multiple antenna elements that together act as a "focusing lens" to form highly directional RF beams. For instance, a 50×50 element phased-array can generate narrow beams of width 3°, providing an antenna gain of 36 dB [7, 14]. However, directional 60 GHz beams are highly susceptible to blockage [5, 7, 15] because of small wavelength, and hence limited ability to diffract around obstacles. When a beam is fully blocked by human body, link budget is penalized by 20 to 30 dB [5, 7]. This is in stark contrast with directional links at lower frequencies [7]. Human movement in a room, therefore, can cause intermittent outage of 60 GHz links. Deployment in [5] observed 1% to 2% of link outage in an environment with 1 to 5 persons, and 14% to 22% with 11 to 15 persons.

The impact of link outage will be amplified at higher network layers. At the transport layer, TCP will respond by timing out, and may even need to re-establish connection [16]. Niche 60 GHz applications such as uncompressed video streaming may suffer from glitches and extended stalling period. Cable-replacement applications such as wireless HDMI, PCIe, USB deem outage event as cable-unplugging [17], and respond to it with very high re-establishment latency. As a result, it degrades the users' quality-of-experience significantly.

2.2 Limitations of Beam Scanning/Searching

A 60 GHz link may overcome blockage by switching Tx's and Rx's beams to form a detour path. Numerous beam-searching protocols, including the default in 802.11ad, has been proposed to search for the best beam pair [6, 18–20] However, any beam-searching protocol bears the following inherent limitations.

Overhead. Although dwelling on each beam direction only takes a few μs (including the time in signal, switching, and RSS feedback), the overhead increases almost quadratically with number of beams ($48.4ms$ for a 16×16 phased-array antenna and $785.7ms$ for a 32×32 one [7]). In addition, beam-searching are usually invoked only at scheduled time-slots, and as such, only responds long after blockage already occurred.

Triggering threshold. Effective triggering of beam-searching remains an open problem. A typical approach is to invoke searching once link SNR changes beyond a threshold [12]. But under such an *aggressive mode*, search and switch may be triggered relentlessly throughout a blockage event, incurring substantial overhead [7]. Under a *conservative mode*, the device defers searching until link outage occurs. It then switches to best available beam pair, if any, to re-establish link. Albeit with less overhead, it may not be able to react quickly to blockage event which leaves the link at outage state for an extended period of time. It is virtually infeasible to set an optimal threshold which depends on elusive human blockage pattern.

Lack of preventive mechanism. Existing beam-searching protocols *react* to blockage, but do not afford any *preventive* mechanism to reduce the likelihood of link outage. An ideal preventive mechanism should be able to assess if a 60 GHz link is robust (*i.e.*, whether it can survive blockage using beam switching), and if not, suggest a re-deployment. However, such assessment is very challenging, because the effectiveness of beam switch-

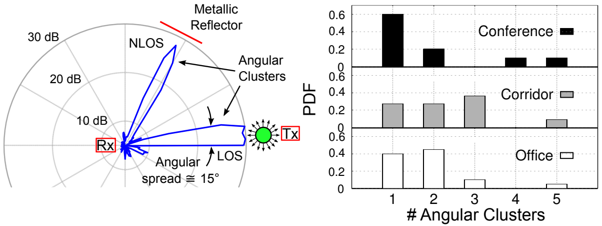


***Figure* 1:** (*a*) *Sparse channel response and strong clustering effect across different AOA.* (*b*) *Distribution of number of strong clusters for 50 links in* 3 *environments.*

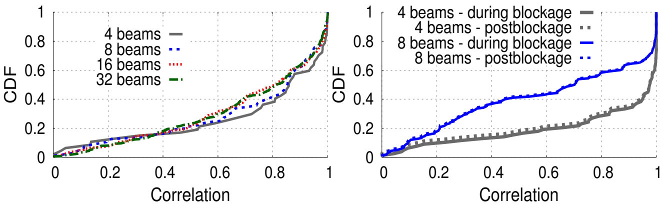


***Figure* 2:** (*a*) *Distribution of correlation between cluster blockage & RSS change of all beams.* (*b*) *Conditional distribution of correlation of RSS change of all other beams w.r.t. strongest beam during & after blockage.*

ing highly depends on elusive environmental factors (reflectors positions, blocking positions *etc.* [7]). Interestingly, our experiments reveal that, maintaining proactive "backup" beams, commonly assumed in simulation models [11–13,21], rarely helps in blockage prevention (Sec. 7). This is due to *spatial correlation*: multiple beams can be blocked simultaneously, even if they seem to point in different physical directions.

3. Measurements and Observations

In this section, we present four measurement observations of 60 GHz channels and phased-array beamforming. These observations constitute the foundation of *BeamSpy*'s prediction framework. Our measurement is conducted using a custom-built 60 GHz software radio platform (detailed in Sec. 6).

***Observation* 1** *Channel sparsity: 60 GHz channels are extremely sparse. The spatial channel response is dominated by a few paths from a few angular directions.*

The sparsity of 60 GHz channel is well known in prior measurement studies [7, 8, 15, 22, 23], and is presented here for completeness. Following a classical channel measurement approach [7,8], we set up an omni-directional 60 GHz transmitter (Tx) in an office environment, while steering a 3° receiver (Rx) to resolve signal paths coming from different spatial angles. Fig. 1(*a*) plots an example *spatial channel profile*, *i.e.*, RSS measured along different Angle-Of-Arrival (AOA). The AOA pattern is extremely sparse — despite the omni-directionality of Tx, the Rx can only receive strong signals from a few densely concentrated directions (referred to as *angular clusters*), each spanning a narrow angle. Such sparsity is because *mmWave signal energy tends to concentrate around the LOS path and a few NLOS paths from strong reflectors*.
We have also measured 50 additional links randomly located in 3 different sites (office, corridor and conference room). Fig. 1(*b*) shows that all the links have 5 or fewer angular clusters, which again corroborates the sparsity.

***Observation* 2** *Spatial correlation: Given a 60 GHz phased-array with multiple beam directions, blockage of one beam affects the performance of other beams.*

For a phased-array receiver beam, the RSS along each spatial angle equals the corresponding beam gain pattern multiplied by the channel gain. For example, under the same setup as in Fig. 1(*a*), Fig. 5(*b*) illustrates the spatial RSS when the receiver's phased-array is tuned to 4 different beam patterns. Note that each beam pattern may contain multiple main lobes and sidelobes, but all 4 beams "share" the same two channel clusters. Therefore, when obstacles block a certain angular cluster, all beams may be affected in a correlated way.

To consolidate this intuition, we examine the impact of blockage on links using omni Tx and phased-array Rx, with 4 to 32 different beam patterns. For each beam, we measure the normalized RSS changes for different blockage positions and stack them into a vector. We then calculate the correlation coefficient between different beams' vectors. Fig. 2(*a*) plots the CDF of pairwise correlation coefficients among 30 randomly deployed links. Around 80% of the links showed a strong inter-beam correlation (coefficient > 0.5), indicating that the *performance of many beams will change in a correlated way even when the blockage seems to land on one beam.*

***Observation* 3** *Blockage invariant correlation: The statistical correlation between different beams is invariant to human blockage.*

Intuitively, whether two beams suffer from correlated performance loss only depends on whether they strongly share certain angular clusters, and the correlation should not be affected by the blockage event itself. For verification, we repeat the previous experiment by blocking only the strongest angular cluster for each link. Meanwhile, we measure the RSS change of all beams. After removing the blockage, we repeat the measurement. Fig. 2(*b*) plots the distribution of the correlation between the performance change of all other beams *w.r.t.* the strongest beam. Evidently, *the spatial correlation between phased-array beams remain unaltered irrespective of blockage events*. *BeamSpy* essentially learns such correlation explicitly using a novel modeling framework (Sec. 4.2.1), and leverages the model to predict the best beam during blockage (Sec. 4.2.2).

***Observation* 4** *Human blockage does not create additional significant angular clusters.*

60 GHz signals are well known to be aquaphobic [7, 24,25]. Therefore, when blocking a 60 GHz link, human

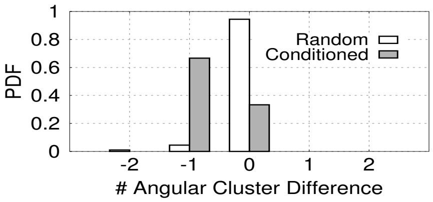


***Figure 3:** Distribution of difference in number of clusters before and during blockage.*

body may absorb most of incoming signals [25], thus partially or completely blocking certain angular cluster(s).

This effect is evident in Fig. 3. Continuing with the prior setup, for each of the random blockages around each link, we identify the strong clusters and plot the PDF of the cluster number difference before and during blockage. A blockage may or may not fully block one cluster, evident from the result that the number of clusters can remain unchanged for more than 90% of the cases. However, if we condition on the cases where blockage landed on one or more clusters, about 63% links show 1 cluster less than those before blockage. Although signals can still bounce off the body, the end effect does not create a new strong/significant cluster.

4. BeamSpy Prediction Framework

Driven by the above observations, *BeamSpy* establishes a beam-quality prediction framework, designed for quasi-stationary 60 GHz links that may be occasionally displaced but frequently blocked by human movement [5]. Fig. 4 illustrates *BeamSpy*'s work flow. When a link is deployed, *BeamSpy* leverages full-beam scanning (such as in 802.11ad) to construct a novel *path skeleton* model, and extrapolate the blockage-invariant spatial correlation between different beams available on the Tx/Rx's phased-arrays. Afterwards, whenever beam quality changes due to blockage (indicated by SNR drop), *BeamSpy* can predict the quality of all other beams by simply measuring the Channel Impulse Response (CIR) of the beam in use.

In what follows, we first provide a primer on how 60 GHz signals are shaped by phased-array beamforming and channel response (Sec. 4.1), based on which we detail *BeamSpy*'s prediction framework (Sec. 4.2 and Sec. 4.3).

4.1 Joint Effect of Phased-array Beamforming and Angle Dependent Channel Distortion

A phased-array can apply a vector of beamforming weights to a set of omni-directional antenna elements to create directional beams, for either a transmitter or a receiver. For ease of exposition, we focus on a 1-D uniformly spaced antenna array, which has N antenna elements and can generate K beam directions/patterns in total. For the k^{th} beam, its *array-factor* (gain at spatial direction θ) is given by [24]:

$$A_k(\theta) = \sum_{n=1}^{N} \mathbf{w}(n,k) \cdot exp(j2\pi ndcos\theta/\lambda) \qquad (1)$$

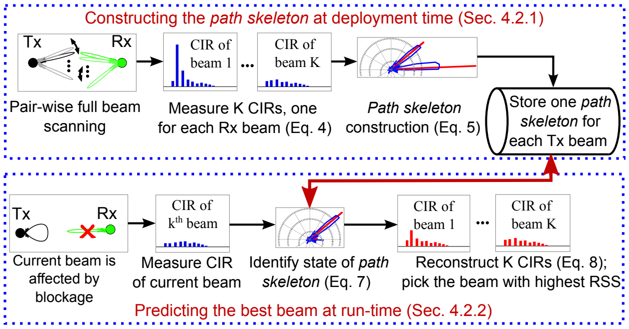

Constructing the *path skeleton* at deployment time (Sec. 4.2.1)
Tx
Rx
CIR of beam 1
...
CIR of beam K
Pair-wise full beam scanning
Measure K CIRs, one for each Rx beam (Eq. 4)
Path skeleton construction (Eq. 5)
Store one *path skeleton* for each Tx beam
Tx
Rx
CIR of k^th beam
CIR of beam 1
...
CIR of beam K
Current beam is affected by blockage
Measure CIR of current beam
Identify state of *path skeleton* (Eq. 7)
Reconstruct K CIRs (Eq. 8); pick the beam with highest RSS
Predicting the best beam at run-time (Sec. 4.2.2)

***Figure 4:** BeamSpy's prediction framework.*

where d is the antenna element spacing and λ the wavelength. **w***(n,k)* is the beamforming weight applied to the n^{th} element when generating beam k. *Each beam corresponds to one antenna gain pattern, with certain spatial directions amplified whereas others weakened.*

For sake of simplicity, suppose the transmitter is omni-directional and receiver is a directional phased-array. When the receiver steers towards the k^{th} beam, its Channel Impulse Response (CIR), h_k, is a joint effect of the *array-factor* in Eq. (1) and the CIR of each antenna element. Suppose Θ is the maximum receiver aperture of each antenna elements (*e.g.*, 360° for omni-directional ones) and P is the number of *paths* the signals can traverse between the transmitter and the receiver. Then, we have:

$$h_k = \sum_{\theta=0}^{\Theta} \sum_{p=1}^{P} A_k(\theta) \cdot \Gamma(p) \cdot \delta(\theta(p) - \theta) \qquad (2)$$

where $\Gamma(p) = \beta(p)e^{j\phi(p)}$ is the channel distortion over path p, and $\beta(p), \phi(p), \theta(p)$ are the attenuation, phase and AOA of signal components traversing along path p. $\delta(\cdot)$ denotes the Dirac delta function capturing the effect of path directions on CIR h_k of k^{th} beam.

Intuitively, the CIR h_k captures the aggregated effect of all paths that arrive at each of the antenna elements, appropriately weighted by **w***(n,k)* and summed together. In practice, not only the receiver, but also the transmitter's phased-array antenna will reshape the channel response, creating directionality effect along different spatial directions. This is equivalent to simply applying the transmitter's array-factor to each path in a reciprocal way.

When steered to beam k, a phased-array receiver can employ the built-in channel-training preambles such as in 802.11ad packets to extract the CIR value h_k, which is required in order to demodulate the packet payload [1].

4.2 Prediction Framework: Model & Algorithm

We formally define a *path skeleton* as *the sparse set of dominating paths that can be used to approximate the spatial channel between a 60 GHz transmitter and receiver.* Note that, the *path skeleton* only depends on the channel and is independent of the beamforming weights at Tx/Rx. However, the *overarching challenge* here is that the Tx/Rx can only measure the CIR when using a given beam, and cannot discriminate the channel distor-

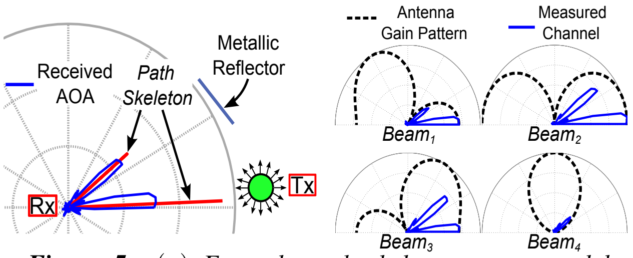


***Figure* 5:** *(a) Example path skeleton constructed by BeamSpy. (b) Measured CIR when receiver's phased-array antenna switches among 4 beam patterns.*

tion/blockage along each specific path. But intuitively, since the channel is sparse, the very few number of dominating paths form a *path skeleton* that determines the performance of all receive beams. The core idea of *BeamSpy* is to "reverse-engineer" the *path skeleton* between the Tx and Rx (Sec. 4.2.1) and, when blockage occurs, estimate the blocked paths within the skeleton, and then predict the CIR of unobserved beams based on their known beamforming weights (Sec. 4.2.2).

4.2.1 Constructing the Path Skeleton

Fig. 4 (top) shows the one-time procedure to construct the *path skeleton*. When the 60 GHz AP and clients are deployed, *BeamSpy* invokes a full beam-searching procedure (such as in 802.11ad) once to capture the pre-blockage CIR between each pair of Tx and Rx beams. For each Tx beam, *BeamSpy* uses an array of K entries to store the CIR of the K receive beams.

Typical 60 GHz phased-array transmitters need to generate both highly directional and quasi-omni-directional beams [1, 7]. For simplicity, we only focus on quasi-omni transmit beam, whereas the receive beam can be any $k \in [1, K]$. *BeamSpy* constructs a *path skeleton* for each transmit beam. Simply put, the *path skeleton* consists of M *paths* arriving at receiver, which can be used to re-model the K CIR entries. These M paths should contain majority of the spatial information of the channel to represent the sparse set of signal clusters (*Observation* 1) between the Tx and the Rx.

So, *how to construct the M paths, given that the receiver can only measure the CIR of each receive beam? BeamSpy* solves this problem using a reverse-engineering model. Following Eq. (2), the CIR of beam k ($\in [1, K]$) can be represented as:

$$\begin{aligned} h_k^{rep} &= \textstyle\sum_{i=1}^{M} A_k(\theta_i) \cdot a_i e^{j\phi_i} \\ \text{and let } H^{rep} &= \left\{h_1^{rep}, h_2^{rep}, \ldots, h_K^{rep}\right\} \end{aligned} \quad (3)$$

Here, each path p_i is represented by a triplet θ_i, a_i, ϕ_i, denoting its angle-of-arrival, amplitude and phase.
Denote the measured CIR of the K beams as:

$$H^{ms} = \left\{h_1^{ms}, h_2^{ms}, \ldots, h_K^{ms}\right\} \quad (4)$$

Then, *BeamSpy* resolves the M paths as:

$$\{\hat{p}_1, \cdots, \hat{p}_M\} = \underset{\{p_1, \cdots, p_M\}}{\operatorname{argmin}} \left|\left|H^{ms} - H^{rep}\right|\right|^2 \quad (5)$$

Said differently, *BeamSpy* searches for M skeleton paths that can reproduce the measured CIR of all K beams with minimum error. This is a non-linear least mean square error curve fitting problem. We solve the problem using the *Levenberg-Marquardt Algorithm (LMA)* [26], which are widely applied in non-linear inverse problems.

When $M > K$, the problem (5) can become underdetermined, since we are fitting a skeleton of M paths to K measured samples. Therefore, *given the receiver owns K beams, BeamSpy uses a skeleton of $M \leq K$ paths to represent the channel*. In practice, $M \leq K$ holds because the number of strong angular clusters (paths) is typically below 5 in 60 GHz indoor/outdoor channels [9, 22] (*c.f.* Fig. 1(b)), whereas $K \geq 8$ even for a small 4-element phased-array [7]. We will further discuss about choice of M with system-level constraints in Sec. 4.2.2. Fig. 5(a) illustrates an example of *path skeleton* constructed using 4 angular CIR patterns, measured when the receiver is tuned to 4 different beam patterns (Fig. 5(b)).

4.2.2 Predicting the Best Beam

Fig. 4 (bottom) shows *BeamSpy*'s run-time prediction procedure. Specifically, given the measured CIR of current beam, *BeamSpy* estimates which paths within the *path skeleton* are affected by blockage. It then immediately predicts the quality of all alternative beams, based on the *a priori path skeleton* which captures the invariant spatial correlation between beams.

To identify the affected paths, *BeamSpy* makes the following approximation, inspired by *Observation* 4: human blockage annihilates existing skeleton paths, but does not create new paths. Suppose, the current receive beam index is k, with measured CIR h_k^{obs}. Let $b_i \in \{0, 1\}$ be an indicator variable denoting whether the skeleton path p_i is blocked. Following Eq. (3), we can represent the current beam's CIR as:

$$h_k^{pred} = \textstyle\sum_{i=1}^{M} A_k(\theta_i) \cdot b_i \cdot a_i e^{j\phi_i} \quad (6)$$

where $b_i \cdot a_i$ models path p_i's amplitude during blockage. Then we fit Eq. (6) to measured CIR h_k^{obs} to estimate $\hat{\mathbf{b}}$, the blocked/non-blocked states of all the skeleton paths:

$$\hat{\mathbf{b}} = \{\hat{b}_1, \cdots, \hat{b}_M\} = \underset{\{b_1, \cdots, b_M\}}{\operatorname{argmin}} \left|\left|h_k^{pred} - h_k^{obs}\right|\right|^2 \quad (7)$$

Finally, given the estimated $\hat{\mathbf{b}}$, and the pre-blockage *path skeleton* (Sec. 4.2.1), *BeamSpy* can reconstruct the CIR of any unobserved beam k' ($k' \neq k$):

$$h_{k'}^{pred} = \textstyle\sum_{i=1}^{M} A_{k'}(\theta_i) \cdot \hat{b}_i \cdot a_i e^{j\phi_i} \quad (8)$$

The reconstructed CIR can be straightforwardly converted to link quality metric, like RSS or effective SNR [27], based on which *BeamSpy* can identify the best beam.
Note that, the *Minimum Mean Square Error* (*MMSE*) between current beam's measured and reconstructed CIR,

$$e_M = \min_{\{b_1, b_2, \cdots, b_M\}} \left|\left|h_k^{pred} - h_k^{obs}\right|\right|^2 \quad (9)$$

characterizes the error when *BeamSpy* re-models blockage using $\hat{\mathbf{b}}$. This is a salient metric, later used as a *confidence level* of *BeamSpy*'s beam quality prediction.

Handling partial blockage and run-time execution. In practice, a path may consist of a cluster of angles, and human body may block only part of the cluster. To account for partial blockage, we quantize the elements $\hat{b}_i \in \hat{\mathbf{b}}$ into Q discretized levels and by default set $Q = 4$, corresponding to levels $\{0, \frac{1}{3}, \frac{2}{3}, 1\}$. When multiple paths are affected concurrently (due to *either single or multi-person blockage*), the prediction framework is still applicable as multiple elements in $\hat{\mathbf{b}}$ may become 0 simultaneously. The non-linear problem in (7) may be solved using *LMA*, similarly to (5). However, the complexity may be too high for run-time execution. We simplify the solution using a *look-up table* approach. During the *path skeleton* construction phase, we build a table of size Q^M, each entry corresponding to the h_k^{pred} for a given configuration of $\hat{\mathbf{b}}$. At run time, to solve the problem (7), *BeamSpy* can simply look for the entry that matches h_k^{obs} with minimum error. To reduce the lookup time, we empirically limit M, the number of skeleton paths, to a maximum of 8. In practice, this empirical choice works because the typical number of angular clusters for 60 GHz indoor/outdoor channels is well below 5 [9, 22].

We emphasize that *BeamSpy* needs no PHY layer modification to the 60 GHz radios. It requires only channel response for each receive beam direction. Today's WiFi drivers already allow access to such information [27] and we expect this trend to continue for the 60 GHz drivers.

4.3 Operations of the Prediction Protocol

Lightweight prediction protocol. The above prediction framework focused on predicting the best Rx beam for a given quasi-omni Tx beam. We now describe how to extend the framework to predict the best *pair* of Tx and Rx beams. This is realized using a lightweight two-step signaling procedure illustrated in Fig. 6(a).

First, the AP temporarily switches to a quasi-omni antenna mode. The client measures the corresponding CIR using its current beam, and predicts the quality of alternative beams using the above algorithm. Then, the client feeds back the index of its new best beam direction (feedback sent via this new direction). Under the client's new beam direction, the AP predicts its new best beam, using the prediction algorithm in a reciprocal way.

The two-step signaling can be sneaked into existing MAC protocols, *e.g.* 802.11ad, as a background procedure. During normal data transmission, AP downgrades to quasi-omni mode, and invoke two-step signaling just like a normal Data-ACK exchange. *BeamSpy*'s prediction framework requires that the AP's quasi-omni beam cover its all possible fine beam directions, thus exciting all paths they can generate. Multiple quasi-omni beams may be invoked separately to repeat the two-step signaling and satisfy this requirement. In practice, a quasi-omni beamwidth can be $60°$ to $180°$, and 1 to 3 quasi-omni beams are sufficient to meet this goal.

Two additional points are worth discussion here:

(1) *BeamSpy* requires that channel estimation is still feasible after blockage. This is reasonable because channel training preamble is much more robust than data payload. For exmaple, 802.11ad uses a training sequence with a spreading factor of 128. Thus, channel estimation is still feasible even if link SNR is 21 dB $(10\log_{10}(128))$ lower than the minimum SNR needed to demodulate data.

(2) In the second signalling step, even if channel estimation is feasible, the client may fail to convey its best beam index to AP, *i.e.*, the signaling packet may be lost and AP times out. However, as long as the AP can extract the client→AP CIR, it can recover the index as follows. It can search through each *path skeleton* corresponding to each of the client's Tx beams, and apply a *minimum Euclidean distance* metric to infer the client's best beam index j. Then, it applies *BeamSpy*'s prediction algorithm as if j is directly fed back by the client.

In practice, *BeamSpy*'s signaling failure rarely occurs, because it can employ a fast beam adaptation protocol (Sec. 5.1) to ensure the link migrates to the best alternative beam pair before the current one is fully blocked.

Refreshing *path skeleton*. After its initial construction, the *path skeleton* is refreshed on demand. *BeamSpy* can instantaneously detect staleness of the *path skeleton* using the *confidence level* parameter e_M (Eq. (9)). It reruns the *path skeleton* construction (Sec. 4.2.1) if the e_M drifts from its initial value beyond a threshold (default to 20%). We note that the triplet θ_i, a_i, ϕ_i that characterize each path p_i are only affected by link distance, Tx/Rx orientation and strong reflectors in the environment. Human movement does not affect the parameters, because the effect is mostly weakening/blocking the paths, not creating new skeleton paths (*Observation* 4). Strong indoor reflectors (*e.g.* concrete walls, large metal furnitures *etc.*) are typically not changed frequently. So *BeamSpy* only needs to refresh the *path skeleton* at a very coarse time scale. We will evaluate the impact of environment change on the *path skeleton* update in Sec. 7.

5. Applications of BeamSpy

BeamSpy can potentially facilitate many higher layer protocols and network management schemes. A higher layer protocol may exploit *BeamSpy*-predicted RSS of beams to jointly adapt bit-rate and beam direction. A 60 GHz mesh network may use *BeamSpy* to instantaneously predict the best relay node that can help detour blockage. Further, network planners may use *BeamSpy* for what-if analysis when deploying 60 GHz links. In this section, we explore two salient applications of *BeamSpy* to enable robust and efficient 60 GHz networking. We use 802.11ad as a baseline MAC/PHY, but *BeamSpy* can be similarly integrated with other standards *e.g.* 802.15.3c.

5.1 Fast Beam Adaptation Under Blockage

We first employ *BeamSpy* to enable efficient beam switching under human movement and blockage. Specifically, *BeamSpy* acts as a meta-protocol to augment 802.11ad, so as to quickly recover from link outage without the high-overhead beam searching [7].

Assuming the AP and client have established association, Fig. 6(b) illustrates the run-time state machine of the link. A legacy 802.11ad link transits between 3 states: *Norm*, *Outage*, and *Scan*. An 802.11ad link responds to the blockage event by triggering beam-scanning that searches for the best alternative beam pair. However, whether to use an aggressive or conservative threshold remains an open problem, due to a tradeoff between overhead and responsiveness (Sec. 2.2). *BeamSpy* overcomes the dilemma using its prediction framework together with the two-step signaling. Similar to aggressive mode, prediction is triggered when SNR changes. But, instead of a full scan, *BeamSpy* only needs to inspect its current beam quality, execute the aforementioned prediction algorithm, and estimate the best alternative Tx/Rx beam pair. Further, *BeamSpy* can help the link make an informed decision while utilizing its prediction result.

Recall that, *BeamSpy* outputs an *MMSE* (e_M) of model fit as *confidence level* of prediction (Sec. 4.2.2). Intuitively, a higher e_M indicates a larger error in estimating the blocked/non-blocked states of *path skeleton*, and thus poor prediction accuracy. We leverage e_M to make a probabilistic decision between utilizing *BeamSpy*'s prediction result and invoking an 802.11ad full-scanning. Specifically, we first use *BeamSpy* to predict the best available beam and evaluate e_M following Eq. (9). e_M is normalized *w.r.t.* to the current beam's channel gain $||h_k^{obs}||^2$, and the normalized value manifests how likely *BeamSpy* predicts wrongly. Then, we set the probability p of utilizing *BeamSpy*'s prediction result as $p = 1 - (e_M/||h_k^{obs}||^2)$. The probabilistic scheme reaps the benefit of accurate prediction, whenever possible, thus evading high overhead of beam-scanning. We will evaluate link performance gain from *BeamSpy*'s fast beam adaptation under intermittent human blockages in Sec. 7.1.2.

5.2 Link Outage Risk Assessment

BeamSpy does not guarantee a blocked link can be recovered via beam switching – such feasibility solely depends on Tx/Rx placement and environmental reflectivity [7]. However, *BeamSpy* can predict how likely a link deployment is to fail completely when blockage occurs. We call this scheme *risk assessment*. Risk assessment is critical when a 60 GHz link is deployed as a fixture, *e.g.*, from ceiling/wall to a furniture in home. Instead of a re-deployment (which may be unaffordable) upon link outage, at deployment time, the user can attempt different placement, orientation, or even place new reflectors, until the risk assessment mechanism indicates a low risk.

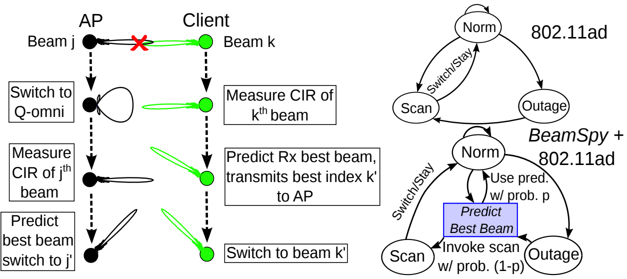


Figure 6: (a) *Two-step signalling procedure (Sec. 4.3).* (b) *Radio state-transition of 802.11ad (top), and BeamSpy's fast beam adaptation (bottom) that augments 802.11ad using the prediction framework.*

Risk assessment may be realized via empirical ways. For instance, one can conduct a war-driving and blocking the LOS to measure the likelihood of link outage. However, this can be tedious and cumbersome, since blocking different LOS positions may or may not fail a 60 GHz link with beam switching capability [7]. Alternatively, one can run an 802.11ad full scan and check if alternative Tx/Rx beam pairs can establish link connection besides the LOS beam pair. However, different beam pairs can suffer from *correlated outage* when blockage occurs (Sec. 3). This is again due to spatial correlation effect which can not be captured by empirical solutions.

With *BeamSpy*, we design a risk assessment mechanism to overcome such fundamental limitations. To capture the correlated blockage effect, we define the risk of a link deployment as a conditional probability,

$$\kappa = \mathbb{P}(\textit{Link outage}|\textit{Beam with highest RSS is blocked})$$

Link outage occurs if no beam sustains the RSS required by the lowest bit rate, even after beam switching.

To evaluate the κ-factor for a given deployment, *BeamSpy* first constructs the *path skeleton* and then "rehearses" all possible blockage patterns over the beam with highest RSS. Each blockage pattern is modeled by a quantized reduction of the amplitude of one or more paths that the highest-RSS beam traverses, and 4 quantization levels are used by default. Then, following Sec. 4.2.2, *BeamSpy* predicts quality of all beams under each possible blockage pattern, and computes deployment risk κ as fraction of cases where no beam can sustain the minimum bit rate. We summarize *BeamSpy*'s risk assessment procedure in Alg. 1. The algorithm is statistical in nature. It does not make any assumption about the human movement pattern or the absolute RSS drop due to blockage.

6. Testbed and Implementation

60 GHz Software-Radio Platform. To implement *Beam-*

Algorithm 1 *Link Outage Risk Assessment*

```
Initialize quantization level: $Q = 4$ ; $C_{Survival} = K \times Q^M$. $K$ = Number of beams, $M$ = Number of paths.
foreach Tx beam
    Rx constructs path skeleton with $M$ paths (Sec. 4.2.1);
    Initialize quantization matrix, $C_M$ with $Q^M$ entries;
    foreach row of $C_M \to q$: $[q_1, q_2, \ldots, q_M]$
        Modulate path skeleton $[p_1, p_2, \ldots, p_M]$ by quantization vector $q$; Re-construct $K$ beams from $M$ modulated paths
        If RSS of all $K$ beams $<$ minimum RSS requirement
            $C_{Survival} = C_{Survival} - 1$;
Deployment risk, $\kappa = 1 - (C_{Survival}/K \cdot Q^M)$;
```

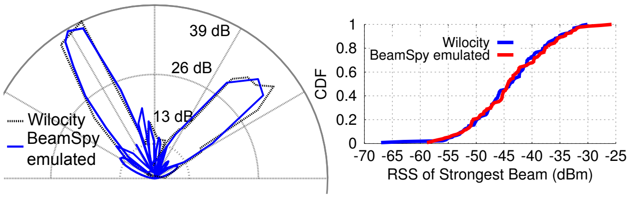


Figure 7: (a) *Two example beam patterns of the Wilocity phased array [30], in comparison with the corresponding emulated beam patterns in our implementation.* (b) *Distribution of RSS of LOS strongest beam in Wilocity and BeamSpy-emulated links.*

Spy and evaluate its efficacy, we use *WiMi* [7], a custom-built 60 GHz software radio platform. *WiMi* allows programmable waveform generation and received-signal processing on a PC host. Its RF front-end operates on 57-64 GHz carrier frequencies with 10 dBm output power and 245.76 Msps baseband sampling rate.

Due to lack of COTS 60 GHz phased-array antennas, we use a trace-driven approach to emulate channel response of a 60 GHz phased-array with a given set of beamforming weights. The foundation of this approach has been well established [6,28]. Simply put, it conducts an angle-wise multiplication between the phased-array's antenna gain, and the spatial channel response (*i.e.*, AOA pattern) between the transmitter and receiver. However, unlike the approach in [6, 28] that simulates the AOA patterns, we follow Rappaport *et al.* [8] to directly measure the AOA pattern of a 60 GHz link. Specifically, we equip *WiMi* with a highly directional 3° horn antenna, which is steered using a programmable real-time motion control system [29]. We measure the fine-grained AOA trace spanning 360°, and then convolve it with a standard 802.11ad beamforming codebook. This emulation approach is applied to each link in our experiment, with different blockage patterns, and different phased-array sizes that produce different number of beams. Note that, the mechanical movement of the horn antennas are used to resolve spatial channel and the movement itself does not affect the channel. Since spatial channel response is obtained by testbed measurement, a real phased-array may differ from emulated one only in its imperfect antenna gain pattern. As a validation, we measure transmitter beam pattern of the Wilocity 60 GHz radio [30] which has a built-in phased-array, and compare with the emulated one. The result (Fig. 7(a)) shows a close-match between these two. Also, we compare RSS of 15 different LOS links for both Wilocity radio (following approach in [31]) and *BeamSpy*-emulated. The *WiMi* transmitter and receiver were colocated with Wilocity for each link. Since, Wilocity radio does not allow us to control its beam direction, we only measure RSS of LOS strongest beam. Fig. 7(b) shows that *BeamSpy*-emulated RSS distribution follows the Wilocity distribution.

Due to hardware constraint, we can only use *WiMi* to send narrow-band (bandwidth=245.76 MHz) signals rather than 802.11ad-compatible one (1.7 GHz). The use of narrowband test equipment does not fundamentally affect our experimental validation, since directional mmWave channels experience minute frequency selectivity [22]. Even under a frequency selective channel, *BeamSpy* can be trivially extended by running prediction over different subchannels separately, and synthesizing the CIR to compute RSS across an entire band.

Finally, we implement *BeamSpy*'s prediction algorithm and applications within the software radios' PC host, on top of an existing library that implements a virtual-clock-driven 802.11ad MAC [7]. The MAC module accurately follows 802.11ad's default timing parameters when enforcing packetization with preamble, beaconing, beam-searching, inter-frame spacing, ACK, *etc.* The receiver measures RSS and noise floor on a per-packet basis, and translates it into achievable bit-rate following an 802.11ad specific rate table similar to the approaches in [7,32].

Emulating Transport/Application Layers. We develop an emulation framework that can replay transport and application layer protocols, on top of fine-grained link-layer traces measured from *WiMi*. Our implementation adapts the popular *Dummynet* emulator kernel [33]. During blockage, 60 GHz link quality (throughput and packet delay/loss) can vary significantly at fine time scales. However, *Dummynet* can only configure link quality over coarse time scales through user space commands. Besides, packet losses can only be introduced probabilistically, which hinders accurate link-layer trace playback. To overcome such limitations, we augment the kernel to emulate link quality as functions of time at a fine granularity of 1 *ms*.

7. Evaluation

7.1 Micro-benchmarks

We now evaluate *BeamSpy*, focusing on three key performance questions: (1) How *accurately* can *BeamSpy* predict beam quality? (2) How much *performance improvement* can *BeamSpy* bring to a 60 GHz link under blockage? (3) How well can *BeamSpy* assess the *risk factor* of arbitrary 60 GHz link deployment?

7.1.1 Prediction Accuracy

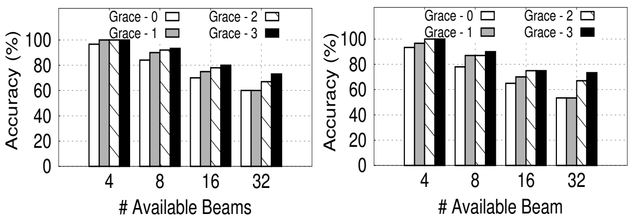


***Figure* 8:** *Accuracy of prediction.* (a) *Best Rx beam under a quasi-omni Tx beam.* (b) *Joint best transmit-receive beam pair. Grace-*n *indicates the oracle best beam lies within first* $(n+1)$ *predicted best ones.*

Accuracy of beam-quality prediction. We test link pairs deployed across 30 locations in an office. By default, a link runs a quasi-omni-directional transmitter. A human body statically blocks at a random position within LOS, and a 3° directional receiver captures AOA trace of the link before and during the blockage. Before blockage, the receiver emulates phased-array beams (Sec. 6) from the AOA trace and constructs the *path skeleton* (Sec. 4.2.1). During blockage, ground-truth RSS of all beams is measured first, but when running *BeamSpy*, the receiver only employs the CIR change of the single beam in use to predict the RSS of all other beams (Sec. 4.2.2).

Fig. 8(a) shows the accuracy of predicting the best receiver beam index for 4 phased-arrays with different sizes (*i.e.*, different number of antenna elements, and hence number of available beams). For a 4-beam receiver, *BeamSpy*'s mean prediction accuracy is well over 90%. The prediction accuracy drops to 71% as the number of available beams increases to 16, and 60% for 32. However, the oracle best beam still falls within the top 4 predicted beams with high probability (73% to 100%). We have also tested *BeamSpy*'s two-way signaling protocol, applicable when both the Tx and Rx use directional phased-arrays (Sec. 4.3). The results (Fig. 8(b)) show a similar level of accuracy as in predicting the best Rx beam.

The imperfectness in *BeamSpy*'s prediction stems from two factors: (1) The entire spatial channel response is represented by only few strong paths, which induces errors as the number of beams increase; (2) Approximating blockage impacts using quantized ampltitude degdration. Despite this imperfectness, *BeamSpy* can substantially improve link robustness by acting as a meta-protocol for fast beam adaptation (Sec. 7.1.2), and as a risk assessment mechanism to guide link deployment (Sec. 7.1.3).

In practice, beam index matters less than quality of the beam *BeamSpy* predicts. Fig. 9(a) shows the absolute RSS difference between the predicted-best and oracle-best beam, which has a mean error of only 1.2 dB and max. 5 dB even for a 16-beam receiver. This implies that *even when BeamSpy predicts wrong best-beam index, the one it predicts does not have significant RSS difference from the oracle best*. Further, Fig. 9(b) plots the CDF of difference between the predicted RSS and measured

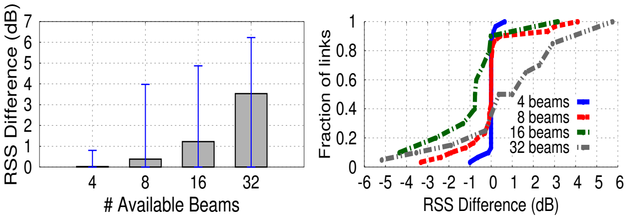


***Figure* 9:** (a) *RSS diff. between predicted and oracle best beam.* (b) *Predicted RSS diff. of the best beam.*

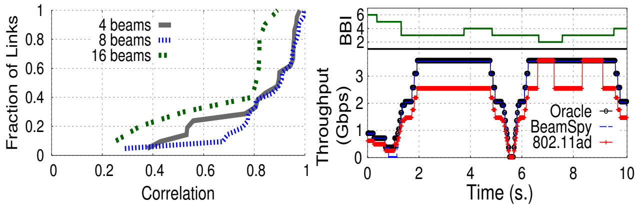


***Figure* 10:** (a) *Correlation between prediction accuracy and* e_M *metric.* (b) *Throughput for 10 s. random walking and blockages. Best Beam Index (BBI) for Rx is shown.*

RSS, focusing on the best beam *BeamSpy* selects. We see that the 93-percentile error stays well within ±4 dB, and median is below 0.8 dB even for the 16-beam case.

Confidence level of *BeamSpy*'s prediction. We evaluate whether the e_M metric in Eq. (9) can effectively indicate the prediction accuracy. We leverage the previous setup and for each blockage case, calculate *BeamSpy*'s prediction accuracy (in finding the best beam), while keeping track of e_M. Then, for each link, we find the correlation coefficient between the measured prediction accuracy and the e_M metric. Fig. 10(a) shows the CDF of correlation result across all links. We observe that for more than 90% of links (with 8 beams), the average correlation is greater than 0.5, Thus, *the* e_M *metric can indeed be used effectively as a confidence level indicator during BeamSpy's prediction*. A closer look to the rest 10% of the links indicates that their performance went below the lowest modulation level under blocakge, and hence almost all beams perform similarly.

Effect of blockage position. We evaluate the prediction accuracy for different blockage positions for 20 link pairs. We setup the links with around $5m$ distance and block them at 10 approximately equal-spaced positions. Fig. 11(a) showcases the result as the position moves from near-Tx to near-Rx. A blockage close to either Tx or Rx blocks all the angular clusters which renders all beams' RSS to drop close to noise level. *BeamSpy* shows relatively lower prediction accuracy there, simply because all beams perform equally poorly. In all other cases, *BeamSpy* shows a high prediction accuracy consistent with our first micro-benchmark. We will evaluate the overall effect of random blockages on *BeamSpy*'s link-layer performance in Sec. 7.1.2.

Effect of quantization. Recall *BeamSpy* employs a quantized *look-up table* to model the effect of blockage (Sec. 4.2.2). Finer quantization helps capture nuances of real

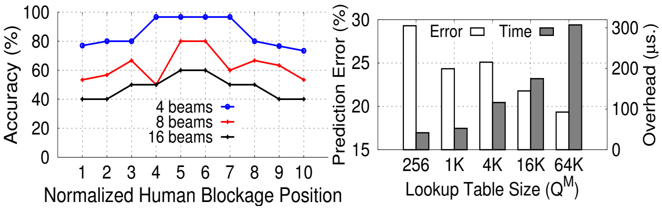


***Figure* 11:** *(a) Accuracy of prediction w.r.t. blockage positions. (b) Effect of increasing path skeleton lookup table size on prediction error and overhead.*

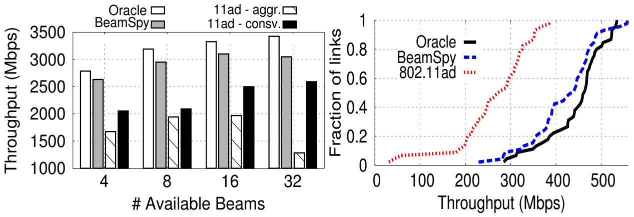


***Figure* 14:** *(a) Performance scaling with different number of available beams in office environment. (b) FTP throughput in office environment.*

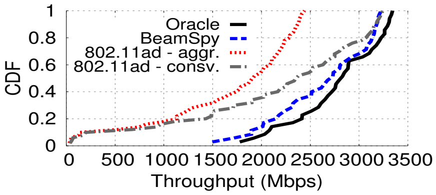


***Figure* 13:** *Throughput distribution of random walking and blockages in office environment.*

blockage effect, but increases the look-up table size and computation overhead. Fig. $11(b)$ illustrates this tradeoff using a 16-beam receiver as example. *BeamSpy*'s prediction framework runs on a desktop PC with 2.6 GHz CPU. With a small table, average prediction error is around 29% and prediction overhead is less than $40\mu s$. As table size grows, the overhead grows proportionally, but the prediction error quickly drops below 20%. We believe a full-fledged optimized firmware/hardware implementation can help minimize *BeamSpy*'s overhead while maintaining its accuracy.

Temporal stability of *path skeleton*. We now evaluate *BeamSpy*'s *path skeleton* refreshment (Sec. 4.3) by generating a set of controlled events around the link. Fig. $12(a)$ shows an example of how the prediction accuracy varies over time when different events occur simultaneously with human blockages. Further, Fig. $12(b)$ quantifies the correlation between *BeamSpy*'s skeleton updates and the event types. Device displacement (mean angular displacement of 10°) or adding a new reflector usually triggers skeleton updates, since they may add new paths into the *path skeleton*. Small changes (involving books, kettle, laptops, chairs, *etc.*) hardly affect the sparse path clusters, and thus do not trigger an update of *BeamSpy*'s *path skeleton*. Overall, we can conclude that *BeamSpy can maintain high prediction accuracy in quasi-stationary environment. It can adapt to infrequent displacement of the Tx/Rx or large reflectors, and is insensitive to small environmental changes.*

7.1.2 Link Performance Gain from Prediction

In this section, we validate fast beam adaptation mechanism enabled by *BeamSpy* (Sec. 5.1). As benchmark comparison, we use 802.11ad with two triggering schemes. Under *aggressive* threshold, beam-scanning is invoked whenever current beam's SNR change leads to bit-rate change; with *conservative* threshold, beam-scanning is only invoked when link cannot sustain the minimum bit-rate. We also compare with an oracle beam adaptation protocol that knows the best beam with no overhead.

Fig. $10(b)$ showcases how one link's throughput and best-beam index varies during random blockages. We see that *BeamSpy*'s throughput closely matches oracle. As a more general test, we repeat this experiment across 50 links deployed in the aforementioned 3 sites. Fig. 13 shows that *BeamSpy* provides a mean throughput gain of 957 Mbps and 479 Mbps (57% & 25%) over 802.11ad with aggressive and conservative thresholds, respectively. *The correctness of the prediction model (Sec. 4.2) depends solely on channel sparsity and beam correlation, which are independent of locations (Sec. 3). Thus, the performance of BeamSpy remains consistently higher than 802.11ad, and close to oracle in different environments.*

Fig. $14(a)$ shows how the link-level throughput scales with the number of available beams. *BeamSpy* performs close to the oracle with small to medium beam number, and 13.2% lower with beam number equal to 32. In all cases, *BeamSpy* outperforms the aggressive 802.11ad by 57–138% and conservative 802.11ad by 25–48%. Interestingly, due to high scanning overhead, the aggressive 802.11ad's performance drops substantially as beam number goes beyond 16, and much worse than the reactive/conservative approach that only responds to disconnections rather than SNR variations.

7.1.3 Performance of Outage Risk Assessment

We leverage the experimental setup in Sec. 7.1.1 to evaluate *BeamSpy*'s risk assessment Alg. 1. To assess the true blockage risk of a link deployment, we repetitively blocks the LOS of a link at random positions, and measure the fraction of cases where no beam can support the SNR needed for the minimum bit-rate. We compare *BeamSpy* with an 802.11ad-based risk assessment [12], which outputs "no risk" if it sees at least one additional backup beam that can establish the link before blockage.

Fig. 15 shows the results. Each dot in the scatter plot compares the *BeamSpy*-predicted risk (κ_p) with the ground-truth risk (κ_m). We observe that *the backup-beam approach largely underestimates the risk.* This effect is amplified in a high RSS regime where many beams can support good quality link before blockage. However, due

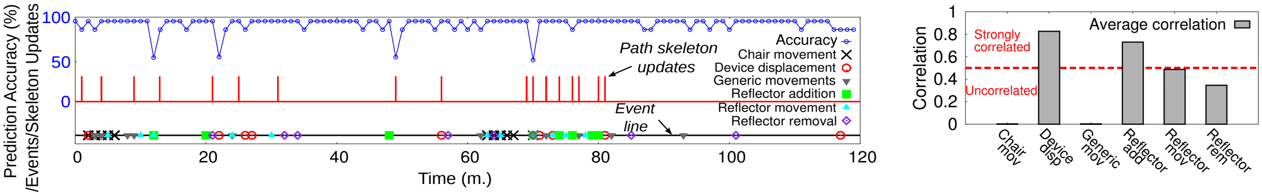


***Figure* 12:** (a) *Time-lapse of prediction accuracy and path skeleton updates under different events.* (b) *Average correlation between updates and event types. Generic movements include moving books, kettle and laptops.*

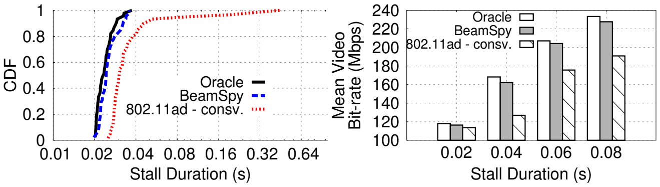


***Figure* 16:** (a) *Distribution of stall durations under fixed video bit-rate.* (b) *Mean achievable video bit-rate conditioned on same stalling rate.*

to *spatial correlation* between beams (*i.e.*, many of them share an "invisible" set of skeleton paths), many of them can be blocked together. *Taking into account the spatial correlation, BeamSpy's risk prediction shows much closer match with the measured risk.* Overall, *BeamSpy*'s predicted risk is slightly biased to the conservative side (on average 14% more risk than the measured one). However, a conservative assessment is more preferable than a overtly optimistic one, since it urges the deployment towards a blockage-proof stage.

7.2 Performance in User-level Applications

To test *BeamSpy* on real applications, we follow the prior setup to collect link rate traces for 50 random walking/blockage near a 8-beam link, and then load the traces into the emulator (Sec. 6). We compare *BeamSpy*'s fast beam adaptation with the conservative 802.11ad, which was shown to outperform the aggressive one.

FTP. We setup an FTP server using the emulator and client downloads a 650 MB file. For each of the downloads, we find mean throughput while a human walks by randomly, and repeat 10 trials. Fig. 14(b) shows resulting FTP throughput distribution, where *BeamSpy*'s median throughput is 147 Mbps (51%) higher than the 802.11ad-aggresive beam-searching.

Uncompressed Video Streaming. We set up a video server that transmits uncompressed video frames over the emulated network stack. The frames are captured and directly displayed through a VLC client. The supported video resolution varies from uncompressed standard definition (640 × 480) to Full-HD (1920 × 1080) at 25 fps. Fig. 16(a) shows that, under fixed Full-HD rate, 802.11ad suffers from high stalling durations during random human walking, and worst case stall can reach $460ms$. In addition, under same median stall-duration, we measure mean video-rate that can be supported (Fig. 16(b)). *BeamSpy* provides 1.3× improvement over 802.11ad. In summary, *BeamSpy simultaneously boosts the video quality while reducing stallings, which can translate into quality-of-experience improvement for end users.*

8. Discussion

Applying *BeamSpy* in outdoor environment. Our experiments have focused only on indoor 60 GHz links that are prone to human blockage. Correctness of *BeamSpy*'s prediction model depends solely on *channel sparsity* and *blockage-invariant spatial correlation*. The former has been extensively demonstrated in both short-range indoor and long-range outdoor environment [10, (Sec. III)]. The latter is a natural consequence of phased-array beamforming in sparse channels. Therefore, *BeamSpy*'s foundation still holds in outdoor environment. We leave the outdoor evaluation as future work.

Asymmetric antenna patterns. In certain 60 GHz deployment, the AP may use a larger phased-array antenna than client, thus creating asymmetric antenna patterns. We note that *BeamSpy* only requires each device to construct its own *path skeleton* to predict the alternate beam quality. Therefore, *BeamSpy* is still applicable under asymmetric phased-array antennas.

Handling client's mobility. *BeamSpy* assumes quasi-stationary link deployment. If the client is mobile, the *path skeleton* construction has to be done at fine time scale, which is as costly as a full beam scanning in 802.11ad. However, by using a wide beamwidth at the mobile client and running *BeamSpy* at a static AP, *BeamSpy*'s *path skeleton* may still hold consistently. We leave the exploration of such mobile scenarios as future work.

***BeamSpy* in multi-links.** A strong interference from co-located links may affect *BeamSpy*'s *path skeleton* construction process and prediction results. However, when multiple devices are served by the same AP, the skeleton construction is already separated in time-domain owing to 802.11ad's MAC protocol. Further, the prediction step is also mutually exclusive as an AP can serve only a single device at a time. Therefore, we expect *BeamSpy* can be easily extended to multi-links setup.

9. Related Work

60 GHz channel and network measurement. Using dedicated channel sounders, existing measurement studies have recognized the unique characteristics of the mmWave channel, especially the significant propagation loss [23] and vulnerability to human blockage indoor [5, 7, 15].

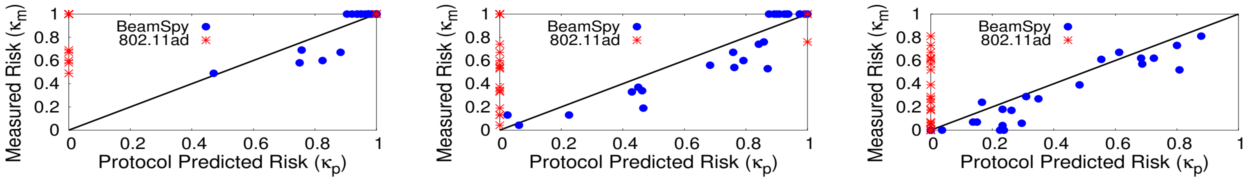


***Figure* 15:** *(a) Performance of BeamSpy's risk assessment algorithm.* (1) *Low RSS regime (< −65 dBm),* (2) *Medium RSS (−65 to −60 dBm),* (3) *High RSS (> −60 dBm).*

Consistent with our observations of channel sparsity, Xu *et al.* [8] and Sur *et al.* [7] showed that even if a transmitter is omni-directional, the received signals tend to be densely concentrated on a few angular clusters. For outdoor pico-cells, the blockage problem is less severe because of higher AP elevation, larger transmit power allowed by regulation and longer link distance [32] (and hence beamwidth expansion).

Efficient beam switching and channel tracking. Despite optimality, brute-force beam scanning incurs high overhead. More efficient beam scanning methods with approximately good performance have been proposed. They exploit hierarchical methods to reduce search space [6, 18, 19, 34]. 802.11ad and 802.15.3c both assimilated the idea, and patched refinement procedures to compensate for the sub-optimality of hierarchical search.

BeamSpy exploits blockage-invariant correlation between beams, a principle that has not been leveraged in previous works. In effect, *BeamSpy*'s fast beam adaptation can be integrated with these beam search protocols, in the same way as it does for 802.11ad (Sec. 5.1). Sparse channel distortion can be represented compactly using compressed sensing [35, 36]. A transmitter can leverage this principle to track the spatial channel response [37]. However, such compressive tracking still requires transmitter to scan multiple beam directions. In contrast, *BeamSpy* deals with link outage due to blockage, and can track the best among all beams by only examining current beam, without brute-force rescanning.

BeamSpy is partly inspired by CSpy [38], which exploited frequency domain correlation between adjacent WiFi channels to predict best channel without probing. Unlike *BeamSpy*, CSpy can only implicitly capture frequency domain correlation through extensive training of a machine learning model.

Surviving blockage. To overcome blockage, beam switching and reflection based methods have been proposed and validated by simulation [11, 12]. But the simulation ignored the practical correlation between beams/paths, which renders the "backup" beams or paths ineffective in practice (Sec. 7). Besides in-band beam searching, out-of-band solutions have been explored, *e.g.*, using a microwave channel [39]. Indirect out-of-band sensing methods like BBS [40] use 2.4 GHz MIMO to determine the spatial channel. Whereas it may help narrow down the search space, it still needs to steer across many 60 GHz beams to find the best-quality narrowest beam. This is because the number of antenna elements in a typical 60 GHz phased-array is much larger than a WiFi array, and thus there can be many narrow beams concentrated within one 60 GHz sector. In addition, like all exisiting beam searching methods, BBS is still a reactive mechanism and incapable of outage risk assessment. Alternatively, a detour path can be formed using relay nodes [14] which suffer less from reflection loss. But the dense deployment incurs additional cost.

Directional antenna networking at lower frequencies. Directional-antenna networking has been extensively studied for the WiFi band, main focus being MAC design in ad-hoc networks [41–44]. More recent work advocates indoor high-speed networking using directional AP/clients [45,46]. Mechanically steerable antennas, and microwave band phased-array antennas have been used [47–49] to maintain directional connectivity with a mobile device. Blockage barely imposes any threat for low-frequency directional links, because of much wider beamwidth (45° or so), and lower penetration loss [7].

Acknowledgment

We thank Jialiang Zhang for help with *WiMi* hardware setup. The work reported in this paper was supported in part by the NSF under Grant CNS-1318292, CNS-1343363, CNS-1350039, CNS-1404613 and CNS-1518728.

10. Conclusion

We have provided experimental evidence of spatial correlation between beams generated by a 60 GHz phased-array antenna. This unique property originates from the fact that the beams can share a sparse set of propagating signal paths. We leverage this observation to design *BeamSpy*, a model-driven framework to predict the performance of multiple beams by inspecting the channel response of a single beam. *BeamSpy*'s modeling parameters are initialized through measurement, and are invariant under link dynamics caused by human blockage. We have validated the feasibility and effectiveness of *BeamSpy* on a 60 GHz testbed, and showcased how it enables efficient and robust 60 GHz networking under human blockage. We believe *BeamSpy* has wider implications for 60 GHz network design than what we have explored in this paper and can benefit a wide range of protocols involving phased-array beamforming.

References

[1] IEEE Standards Association, "IEEE Standards 802.11ad-2012: Enhancements for Very High Throughput in the 60 GHz Band," 2012.

[2] ——, "IEEE Standards 802.15.3c-2009: Millimeter-wave-based Alternate Physical Layer Extension," 2009.

[3] ECMA International, "Standard ECMA-387: High Rate 60 GHz PHY, MAC and PALs," 2010.

[4] T. Rappaport, S. Sun, R. Mayzus, H. Zhao, Y. Azar, K. Wang, G. Wong, J. Schulz, M. Samimi, and F. Gutierrez, "Millimeter Wave Mobile Communications for 5G Cellular: It Will Work!" *IEEE Access*, vol. 1, 2013.

[5] S. Collonge, G. Zaharia, and G. Zein, "Influence of the Human Activity on Wide-Band Characteristics of the 60 GHz Indoor Radio Channel," *IEEE Trans. on Wireless Comm.*, vol. 3, no. 6, 2004.

[6] B. Li, Z. Zhou, W. Zou, X. Sun, and G. Du, "On the Efficient Beam-Forming Training for 60GHz Wireless Personal Area Networks," *IEEE Transactions on Wireless Communications*, vol. 12, no. 2, 2013.

[7] S. Sur, V. Venkateswaran, X. Zhang, and P. Ramanathan, "60 GHz Indoor Networking through Flexible Beams: A Link-Level Profiling," in *Proc. of ACM SIGMETRICS*, 2015.

[8] H. Xu, V. Kukshya, and T. Rappaport, "Spatial and Temporal Characteristics of 60-GHz Indoor Channels," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 3, 2002.

[9] P. Smulders, "Exploiting the 60 GHz Band for Local Wireless Multimedia Access: Prospects and Future Directions," *IEEE Communications Magazine*, vol. 40, no. 1, 2002.

[10] T. S. Rappaport, E. Ben-Dor, J. N. Murdock, and Y. Qiao, "38 GHz and 60 GHz angle-dependent propagation for cellular and peer-to-peer wireless communications," in *IEEE ICC*, 2012.

[11] Z. Genc, U. Rizvi, E. Onur, and I. Niemegeers, "Robust 60 GHz Indoor Connectivity: Is It Possible with Reflections?" in *IEEE Vehicular Technology Conference (VTC-Spring)*, 2010.

[12] X. An, C.-S. Sum, R. Prasad, J. Wang, Z. Lan, J. Wang, R. Hekmat, H. Harada, and I. Niemegeers, "Beam Switching Support to Resolve Link-Blockage Problem in 60 GHz WPANs," in *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2009.

[13] H. Zhang, C. Wu, X. Cui, T. A. Gulliver, and H. Zhang, "Low Complexity Codebook-Based Beam Switching for 60 GHz Anti-Blockage Communication," *Journal of Communications*, vol. 8, no. 7, 2013.

[14] S. Singh, F. Ziliotto, U. Madhow, E. M. Belding, and M. Rodwell, "Blockage and Directivity in 60 GHz Wireless Personal Area Networks," *IEEE JSAC*, vol. 27, no. 8, 2009.

[15] C. Anderson and T. Rappaport, "In-Building Wideband Partition Loss Measurements at 2.5 and 60 GHz," *IEEE Transactions on Wireless Communications*, vol. 3, no. 3, 2004.

[16] C.-Y. Huang and P. Ramanathan, "Network Layer Support for Gigabit TCP Flows in Wireless Mesh Networks," *IEEE Transactions on Mobile Computing*, 2014.

[17] "Multi-gigabit, Low latency connectivity," http://www.wi-fi.org/discover-wi-fi/wigig-certified, 2016.

[18] J. Wang, Z. Lan, C. woo Pyo, T. Baykas, C.-S. Sum, M. Rahman, J. Gao, R. Funada, F. Kojima, H. Harada, and S. Kato, "Beam Codebook Based Beamforming Protocol for Multi-Gbps Millimeter-Wave WPAN Systems," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 8, 2009.

[19] Y. Tsang, A. Poon, and S. Addepalli, "Coding the Beams: Improving Beamforming Training in mmWave Communication System," in *IEEE Global Telecommunications Conference (GLOBECOM)*, 2011.

[20] K. Hosoya, N. Prasad, K. Ramachandran, N. Orihashi, S. Kishimoto, S. Rangarajan, and K. Maruhashi, "Multiple Sector ID Capture (MIDC): A Novel Beamforming Technique for 60-GHz Band Multi-Gbps WLAN/PAN Systems," *IEEE Transactions on Antennas and Propagation*, vol. 63, no. 1, 2015.

[21] B. Gao, Z. Xiao, C. Zhang, L. Su, D. Jin, and L. Zeng, "Double-link beam tracking against human blockage and device mobility for 60-GHz WLAN," in *IEEE Wireless Communications and Networking Conference*, 2014.

[22] T. Rappaport, F. Gutierrez, E. Ben-Dor, J. Murdock, Y. Qiao, and J. Tamir, "Broadband Millimeter-Wave Propagation Measurements and Models Using Adaptive-Beam Antennas for Outdoor Urban Cellular Communications," *IEEE Transactions on Antennas and Propagation*, vol. 61, no. 4, 2013.

[23] P. F. M. Smulders, "Statistical Characterization of 60-GHz Indoor Radio Channels," *IEEE Transactions on Antennas and Propagation*, vol. 57, no. 10, 2009.

[24] T. S. Rappaport, R. W. H. Jr., R. C. Daniels, and J. N. Murdock, *Millimeter Wave Wireless Communications*. Prentice Hall, 2014.

[25] S. Alekseev, A. Radzievsky, M. Logani, and M. Ziskin, "Millimeter Wave Dosimetry of Human Skin," in *Bioelectromagnetics*, 2008.

[26] C.-T. Kim, J.-J. Lee, and H. Kim, "Variable Projection Method and Levenberg-Marquardt Algorithm for Neural Network Training," in *IEEE Industrial Electronics (IECON)*, 2006.

[27] D. Halperin, W. Hu, A. Sheth, and D. Wetherall, "Predictable 802.11 Packet Delivery from Wireless Channel Measurements," in *Proc. of ACM SIGCOMM*, 2010.

[28] M. Park and P. Gopalakrishnan, "Analysis on Spatial Reuse and Interference in 60-GHz Wireless Networks," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 8, 2009.

[29] "Axis360 Motion Control System," http://cinetics.com/two-axis360/.

[30] "Wilocity 802.11ad Multi-Gigabit Wireless Chipset," http://wilocity.com, 2013.

[31] T. Nitsche, G. Bielsa, I. Tejado, A. Loch, and J. Widmer, "Boon and Bane of 60 GHz Networks: Practical Insights into Beamforming, Interference, and Frame Level Operation," in *Proc. of ACM CoNEXT*, 2015.

[32] Y. Zhu, Z. Zhang, Z. Marzi, C. Nelson, U. Madhow, B. Y. Zhao, and H. Zheng, "Demystifying 60GHz Outdoor Picocells," in *Proc. of ACM MobiCom*, 2014.

[33] M. Carbone and L. Rizzo, "Dummynet revisited," in *ACM SIGCOMM Computer Communication Review*, 2010.

[34] K. Ramachandran, N. Prasad, K. Hosoya, K. Maruhashi, and S. Rangarajan, "Adaptive Beamforming for 60 GHz Radios: Challenges and Preliminary Solutions," in *ACM mmCom*, 2010.

[35] W. Bajwa, J. Haupt, A. Sayeed, and R. Nowak, "Compressed Channel Sensing: A New Approach to Estimating Sparse Multipath Channels," *Proceedings of the IEEE*, vol. 98, no. 6, 2010.

[36] C. Berger, Z. Wang, J. Huang, and S. Zhou, "Application of Compressive Sensing to Sparse Channel Estimation," *IEEE Communications Magazine*, vol. 48, no. 11, 2010.

[37] D. Ramasamy, S. Venkateswaran, and U. Madhow, "Compressive Tracking With 1000-Element Arrays: A Framework for Multi-Gbps mm Mave Cellular Downlinks," in *Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2012.

[38] S. Sen, B. Radunovic, J. Lee, and K.-H. Kim, "CSpy: Finding the Best Quality Channel Without Probing," in *Proc. of ACM MobiCom*, 2013.

[39] H. Singh, J. Hsu, L. Verma, S. Lee, and C. Ngo, "Green Operation of Multi-Band Wireless LAN in 60 GHz and 2.4/5 GHz," in *IEEE Consumer Communications and Networking Conference (CCNC)*, 2011.

[40] T. Nitsche, A. B. Flores, E. W. Knightly, and J. Widmer, "Steering with Eyes Closed: mm-Wave Beam Steering without In-Band Measurement," in *Proc. of IEEE INFOCOM*, 2015.

[41] O. Bazan and M. Jaseemuddin, "A Survey On MAC Protocols for Wireless Adhoc Networks with Beamforming Antennas," *IEEE Communications Surveys and Tutorials*, vol. 14, no. 2, 2012.

[42] R. R. Choudhury and N. H. Vaidya, "Deafness: A MAC Problem in Ad-Hoc Networks when using Directional Antennas," in *IEEE ICNP*, 2004.

[43] R. R. Choudhury, X. Yang, R. Ramanathan, and N. Vaidya, "Using Directional Antennas for Medium Access Control in Ad Hoc Networks," in *Proc. of ACM MobiCom*, 2002.

[44] R. Choudhury, X. Yang, R. Ramanathan, and N. Vaidya, "On Designing MAC Protocols for Wireless Networks Using Directional Antennas," *IEEE Transactions on Mobile Computing*, vol. 5, no. 5, 2006.

[45] M. Takai, J. Martin, R. Bagrodia, and A. Ren, "Directional Virtual Carrier Sensing for Directional Antennas in Mobile Ad Hoc Networks," in *Prof. of ACM MobiHoc*, 2002.

[46] X. Liu, A. Sheth, M. Kaminsky, K. Papagiannaki, S. Seshan, and P. Steenkiste, "DIRC: Increasing Indoor Wireless Capacity Using Directional Antennas," in *Proc. of ACM SIGCOMM*, 2009.

[47] A. Amiri Sani, L. Zhong, and A. Sabharwal, "Directional Antenna Diversity for Mobile Devices: Characterizations and Solutions," in *Proc. of ACM MobiCom*, 2010.

[48] C.-F. Shih and R. Sivakumar, "FastBeam: Practical Fast Beamforming for Indoor Environments," in *International Conference on Computing, Networking and Communications (ICNC)*, 2014.

[49] V. Navda, A. P. Subramanian, K. Dhanasekaran, A. Timm-Giel, and S. Das, "MobiSteer: Using Steerable Beam Directional Antenna for Vehicular Network Access," in *Proc. of ACM MobiSys*, 2007.

Compiling Path Queries

Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker
Princeton University

Abstract

Measuring the flow of traffic along network paths is crucial for many management tasks, including traffic engineering, diagnosing congestion, and mitigating DDoS attacks. We introduce a declarative query language for efficient path-based traffic monitoring. Path queries are specified as regular expressions over predicates on packet locations and header values, with SQL-like "groupby" constructs for aggregating results anywhere along a path. A run-time system compiles queries into a deterministic finite automaton. The automaton's transition function is then partitioned, compiled into match-action rules, and distributed over the switches. Switches stamp packets with automaton states to track the progress towards fulfilling a query. Only when packets satisfy a query are the packets counted, sampled, or sent to collectors for further analysis. By processing queries in the data plane, users "pay as they go", as data-collection overhead is limited to exactly those packets that satisfy the query. We implemented our system on top of the Pyretic SDN controller and evaluated its performance on a campus topology. Our experiments indicate that the system can enable "interactive debugging"—compiling multiple queries in a few seconds—while fitting rules comfortably in modern switch TCAMs and the automaton state into two bytes (e.g., a VLAN header).

1 Introduction

Effective traffic-monitoring tools are crucial for running large networks—to track a network's operational health, debug performance problems when they inevitably occur, account and plan for resource use, and ensure that the network is secure. Poor support for network monitoring and debugging can result in costly outages [5].

The network operator's staple measurement toolkit is well-suited to monitoring traffic at a single location (*e.g.,* SNMP/RMON, NetFlow, and wireshark), or probing an end-to-end path at a given time (*e.g.,* ping and traceroute). However, operators often need to ask questions involving packets that traverse specific *paths, over time*: for example, to measure the traffic matrix [19], to resolve congestion or a DDoS attack by determining the ingress locations directing traffic over a specific link [18, 55], to localize a faulty device by tracking how far packets get before being dropped, and to take corrective action when packets evade a scrubbing device (even if transiently).

Answering such questions requires measurement tools that can analyze packets based both on their *location* and *headers*, attributes which may change as the packets flow through the network. The key measurement challenge is that, *in general*, it is hard to determine a packet's upstream or downstream path or headers. Current approaches either require inferring flow statistics by "joining" traffic data with snapshots of the forwarding policy, or answer only a small set of predetermined questions, or collect much more data than necessary (§2).

In contrast, when operators want to measure path-level flows in an network, they should be able to specify concise, network-wide *declarative queries* that are

1. independent of the forwarding policy,
2. independent of other concurrent measurements, and
3. independent of the specifics of network hardware.

The measurements themselves should be carried out by a *run-time system*, that enables operators to

4. get accurate measurements directly, without having to "infer" results by joining multiple datasets,
5. have direct control over measurement overhead, and
6. use standard match-action switch hardware [8, 34].

A Path Query Language. We have developed a *query language* where users specify regular expressions over boolean conditions on packet location and header contents. To allow concise queries over disjoint subsets of packets, the language includes an SQL-like "groupby" construct that aggregates query results anywhere along a path. Different actions can be taken on a packet when

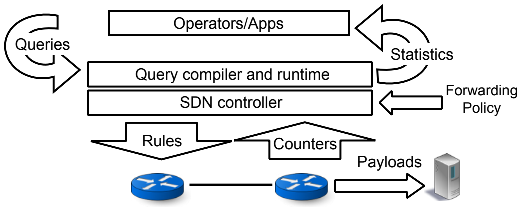


Figure 1: Path Query System.

it satisfies a query, such as incrementing counters, directing traffic to a mirroring port or controller, or sampling at a given rate. These actions may be applied either before or after the packets traverse the matching trajectory.

The Run-time System. To implement a path query, the run-time system programs the switches to record path information in each packet as it flows through the data plane. While prior approaches have tracked packet paths this way [28, 49, 55], a naive encoding of every detail of the path—location and headers—would incur significant overheads. For example, encoding a packet's source and destination MAC addresses, and connection 5-tuple (24 bytes) at each hop incurs more than a 10% space overhead on a 1500-byte packet, if the packet takes six hops.

Instead, we customize packet path information to the input queries. More specifically, the run-time system compiles queries into a deterministic finite automaton (DFA), whose implementation is then distributed across the switches. The state of the DFA is stored in each packet as updated as it traverses the network. Upon receiving a packet, the switch reads the current DFA state, checks conditions implied by the query, writes a new DFA state on to the packet, executes actions associated with forwarding policy, and sends the packet on its way. Further, if a packet reaches an accepting state of the DFA, the actions associated with the accepting state are triggered. Hence, if the action associated with an accepting state is to send the packet to a collector, only packets actually matching a query are ever sent to a collector.

The mechanism we propose has an attractive "pay for what you query" cost model. Intuitively, our technique acts as an application-specific compression scheme for packet content and paths: rather than coding every detail of the packet trajectory, *only the information necessary to answer queries* is represented in the automaton state. When a packet hits an accepting state, all user-requested information about the packet path can be reconstructed.

Prototype Implementation and Evaluation. We have implemented a prototype of our query system on the Pyretic SDN controller [36] with the NetKAT compiler [58]. Our compilation algorithms generate rules both for single and multi-stage match-action tables (*e.g.,* OpenFlow [34], [8]), and we implemented several compiler optimizations that reduce rule-space overhead and query compile time significantly with multi-stage tables. Our system design satisfies requirements (1)-(6) outlined earlier. On an emulated Stanford network topology, our prototype can compile several queries we tested (together) in under 10 seconds. We believe such compile times can enable "interactive" network debugging by human operators. The amount of packet state is less than two bytes, and fits in standard fields like VLAN or MPLS headers. Further, the emitted data plane rules—numbering a few hundreds—fit comfortably in the TCAM available on modern switches [8, 14, 25].

Contributions. In summary, this paper contributes:

1. the design of a query language that allows users to identify packets traversing a given set of paths (§3),
2. an evaluation of query expressiveness and the debugging model through examples (§4),
3. a run-time system that compiles queries to data-plane rules that emulate a distributed DFA (§5),
4. a set of optimizations that reduce query compile time by several orders of magnitude (§6), and
5. a prototype implementation and evaluation with the Pyretic SDN controller and Open vSwitch (§7).

We have open-sourced our prototype [65] and instructions to reproduce the results are available online [46].

Our preliminary workshop paper [38] on designing a path query system was only partly implemented, and the compilation strategy was prohibitively expensive for even moderately-sized networks. In this paper, we implement and evaluate a full system, and develop optimizations essential to make the system work in practice.

2 Design of Path Measurement

How do we know which path a packet took through the network? How do we collect or count all packets going through a specific path? A number of prior approaches [1, 16, 23, 30, 31, 49, 55, 59, 64, 73, 75] aim to answer these questions, but fall short of our requirements.

2.1 Existing Approaches

Policy checking. Approaches like header space analysis [30] and VeriFlow [31] can predict the packets that *could* satisfy certain conditions (*e.g.,* reachability) according to the network's control-plane policy. However, actual data-plane behavior can be different due to congestion, faults, and switch misconfigurations.

'Out-of-band' path measurement. These techniques collect observations of packets from network devices, and *infer* path properties of interest—for example, from independent packet samples (NetFlow [1], [52]), trajectory labels [16], postcards [23], or matched and mirrored

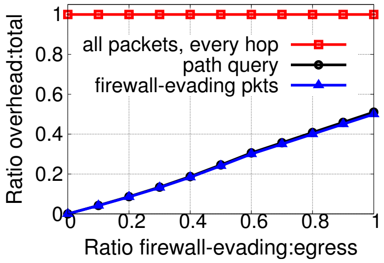


Figure 2: Overheads are limited to traffic matching a query.

packets (wireshark [68], Gigascope [13], [69, 75]). Unfortunately, it is difficult to determine the full path of a single packet through observations spread out in space and time *correctly and efficiently*, for the reasons below.

(i) Dynamic forwarding policies: A simple way to get path measurements is to capture traffic entering a network (*e.g.,* NetFlow [1]) and use the routing tables to estimate the paths the traffic would take. However, packet forwarding changes often due to topology changes, failover mechanisms (*e.g.,* MPLS fast re-route), and traffic engineering. Further, today's devices do not provide the timestamps at which the forwarding tables were updated, so it is difficult to reconcile packet-forwarding state with collected traffic data.

(ii) Packets dropped in flight: It is tricky to estimate actual packet trajectories even when packet forwarding is static. Packets may be dropped downstream from where they are observed, *e.g.,* due to congestion or faulty equipment, so it is difficult to know if a packet actually *completed* its inferred downstream trajectory.

(iii) Ambiguous upstream path: The alternative of observing traffic deeper in a network, on internal links of interest, cannot always tell where the traffic entered. For example, packets with identical header fields may arrive at multiple ingress points, *e.g.,* when packet headers are spoofed as in a DDoS attack, or when two ISPs peer at multiple points. Such packets would follow different paths eventually merging on the same downstream interface: disambiguating them at that point is impossible.

(iv) Packets modified in flight: Compounding the difficulty, network devices may modify the header fields of packets in flight, *e.g.,* NAT. "Inverting" packet modifications to compute the upstream trajectory is inherently ambiguous, as the upstream packet could have contained arbitrary values on the rewritten fields. Computing all possibilities is computationally difficult [74]. Further, packet modifications thwart schemes like trajectory sampling [16] that hash on header fields to sample a packet at each hop on its path.

(v) Opaque multi-path routing: Switch features like equal cost multi-path (ECMP) routing are currently implemented through hardware hash functions which are closed source and vendor-specific. This confounds techniques that attempt to infer downstream paths for packets. This is not a fundamental limitation (*e.g.,* some vendors may expose hash functions), but a pragmatic one.

(vi) High data collection overhead: Since both upstream and downstream trajectory inference is inaccurate, we are left with the option of collecting packets or digests at every hop [23, 59]. However, running taps at every point in the network and collecting all traffic is infeasible due to the bandwidth and data collection overheads. Even targeted data collection using wireshark [68] or match-and-mirror solutions [69, 75] cannot sustain the bandwidth overheads to collect all traffic affected by a problem. Sampling the packets at low rates [16] would make such overheads manageable, but at the expense of losing visibility into the (majority) unsampled traffic. This lack of visibility hurts badly when diagnosing problems for specific traffic (*e.g.,* a specific customer's TCP connections) that the sampling missed.

'In-band' path measurement: These approaches tag packets with metadata to enable switches to *directly* identify packet paths [28, 32, 38, 55, 64, 73]. However, current approaches have multiple drawbacks:

(vii) Limited expressiveness: IP record route [49], traceback [55] and path tracing [64, 73] can identify the network interfaces traversed by packets. However, operators also care about packet *headers*, including modifications to header fields in flight—*e.g.,* to localize a switch that violates a network slice isolation property [30]. Further, the accuracy and overhead of these approaches cannot be customized to requirement: traceback can only accurately record a few waypoints, while path tracing always incurs tag space to record the entire path.

(viii) Strong assumptions: Current approaches require strong assumptions: *e.g.,* symmetric topology [64], no loops [64, 73], stable paths to a destination [55], or requiring that packets reach the end hosts [28, 32]. Unfortunately, an operator may be debugging the network exactly when such conditions *do not* hold.

2.2 Our Approach

We design an accurate "in-band" path measurement system without the limitations of the prior solutions. A runtime system compiles *modular*, *declarative path queries* along with the network's forwarding policy (specified and changing independently), generating the switch-level rules that process *exactly* the packets matching the queries, in operator-specified ways—*e.g.,* counting, sampling, and mirroring. Hence, our system satisfies requirements (1)-(6) laid out in §1. Further, since the emitted data-plane rules process packets at every hop, our system overcomes problems (i), (ii), (iii), and (v) in §2.1. Identifying packet paths "in-band" with packet state untouched by regular forwarding actions removes ambiguities from packet modification (iv), and avoids unneces-

```
field ::= location | header
location ::= switch | inport | outport
header ::= srcmac | dstmac | srcip | dstip | ...
pred ::= true | false | field=value
         | pred & pred | (pred | pred) | ~pred
         | ingress() | egress()
atom ::= in_atom(pred) | out_atom(pred)
         | in_out_atom(pred, pred)
         | in_group(pred, [header])
         | out_group(pred, [header])
         | in_out_group(pred, [header],
                        pred, [header])
path ::= atom | path ^ path | (path | path)
         | path* | path & path | ~path
```

Figure 3: Syntax of path queries.

sary collection overheads (vi). Finally, our query language and implementation allow waypoint and header-based path specification (vii) and do not require strong operational assumptions to hold (viii).

As a demonstration of our query system, Fig. 2 shows that *only* those packets evading a firewall switch in the network core are collected at the network egress, on an emulated Stanford campus topology [2]. In comparison, common alternatives like wireshark will need to collect all network traffic to reliably catch such packets.

Our system must overcome the challenges below.

(i) Resource constraints: The space to carry packet trajectory metadata is limited, as packets must fit within the network's MTU. Further, switch rule-table space is limited [14], so the system should generate a compact set of packet-processing rules. Finally, to be usable for operator problem diagnosis, the system should compile queries in an acceptable amount of time.

(ii) Interactions between multiple measurement and forwarding rules: Switches must identify packets on all operator-specified paths—with some packets possibly on multiple queried paths simultaneously. The switch rules that match and modify packet trajectory metadata should not affect regular packet forwarding in the network, even when operators specify that packets matching the queries be handled differently than the regular traffic.

Practically, our query system is complementary to other measurement tools which are "always on" at low overheads [1, 52, 75]—as opposed to completely replacing those tools. Instead, our query system enables operators to focus their attention and the network's limited resources on clearly-articulated tasks during-the-fact.

3 Path Query Language

A path query identifies the set of packets with particular header values and that traverse particular locations. Such queries can identify packets with changing headers, as happens during network address translation, for instance. When the system recognizes that a packet has satisfied a query, any user-specified action may be applied to that packet. Fig. 3 shows the syntax of the language. In what follows, we explain the details via examples.

Packet Predicates and Simple Atoms. One of the basic building blocks in a path query is a *boolean predicate* (`pred`) that matches a packet at a single location. Predicates may match on standard header fields, such as:

```
srcip=10.0.0.1 & dstip=10.0.0.2
```

as well as the packet's location (a switch and interface). The predicates `true` and `false` match all packets, and no packets, respectively. Conjunction (&), disjunction (|), and negation (∼) are standard. The language also provides syntactic sugar for predicates that depend on topology, such as `ingress()`, which matches all packets that enter the network at some *ingress* interface, *i.e.*, an interface attached to a host or a device in another administrative domain. Similarly, `egress()` matches all packets that exit the network at some *egress* interface.

Atoms further refine the meaning of predicates, and form the "alphabet" for the language of path queries. The simplest kind of atom is an `in_atom` that tests a packet as it *enters* a switch (*i.e.*, before forwarding actions). Analogously, an `out_atom` tests a packet as it *leaves* the switch (*i.e.*, after forwarding actions). The set of packets matching a given predicate at switch entry and exit may be different from each other, since a switch may rewrite packet headers, multicast through several ports, or drop the packet entirely. For example, to capture all packets that enter a device `S1` with a destination IP address (say `192.168.1.10`), we write:

```
in_atom(switch=S1 & dstip=192.168.1.10)
```

It is also possible to combine those ideas, testing packet properties on both "sides" of a switch. More specifically, the `in_out_atom` tests one predicate as a packet enters a switch, and another as the packet exits it. For example, to capture all packets that enter a NAT switch with the virtual destination IP address `192.168.1.10` and exit with a private IP address `10.0.1.10`, we would write:

```
in_out_atom(switch=NAT & dstip=192.168.1.10,
            dstip=10.0.1.10)
```

Partitioning and Indexing Sets of Packets. It is often useful to specify groups of related packets concisely in one query. We introduce *group atoms*—akin to SQL `groupby` clauses—that aggregate results by packet location or header field. These group atoms provide a concise notation for partitioning a set of packets that match a predicate in to subsets based on the value of a particular packet attribute. More specifically, `in_group(pred,`

Example	Query code	Description
A simple path	`in_atom(switch=S1) ^ in_atom(switch=S4)`	Packets going from switch S1 to S4 in the network.
Slice isolation	`true* ^ (in_out_atom(slice1, slice2) | in_out_atom(slice2, slice1))`	Packets going from network slice slice 1 to slice2, or vice versa, when crossing a switch.
Firewall evasion	`in_atom(ingress()) ^ (in_atom(∼switch=FW))* ^ out_atom(egress())`	Catch packets evading a firewall device FW when moving from any network ingress to egress interface.
DDoS sources	`in_group(ingress(), [switch]) ^ true* ^ out_atom(egress(), switch=vic)`	Determine traffic contribution by volume from all ingress switches reaching a DDoS victim switch vic.
Switch-level traffic matrix	`in_group(ingress(), [switch]) ^ true* ^ out_group(egress(), [switch])`	Count packets from any ingress to any egress switch, with results grouped by (ingress, egress) switch pair.
Congested link diagnosis	`in_group(ingress(), [switch]) ^ true* ^ out_atom(switch=sc) ^ in_atom(switch=dc) ^ true* ^ out_group(egress(), [switch])`	Determine flows (switch sources → sinks) utilizing a congested link (from switch sc to switch dc), to help reroute traffic around the congested link.
Port-to-port traffic matrix	`in_out_group(switch=s, true, [inport], [outport])`	Count traffic flowing between any two ports of switch s, grouping the results by the ingress and egress interface.
Packet loss localization	`in_atom(srcip=H1) ^ in_group(true, [switch]) ^ in_group(true, [switch]) ^ out_atom(dstip=H2)`	Localize packet loss by measuring per-path traffic flow along each 4-hop path between hosts H1 and H2.

Table 1: Some example path query applications. Further examples can be found in an extended version [39].

[h1,h2,...,hn]) collects packets that match the predicate pred at switch ingress, and then divides those packets into separate sets, one for each combination of the values of the headers h1, h2, ..., hn. The out_group atom is similar. For example,

```
in_group(switch=10, [inport])
```

captures all packets that enter switch 10, and organizes them into sets according to the value of the inport field. Such a groupby query is equivalent to writing a series of queries, one per inport. The path query system conveniently expands groupbys for the user and manages all the results, returning a table indexed by inport.

The in_out_group atom generalizes both the in_group and the out_group. For example,

```
in_out_group(switch=2, [inport], true, [outport])
```

captures all packets that enter switch=2, and exit it (*i.e.*, not dropped), and groups the results by the combination of input and output ports. This single query is shorthand for an in_out_atom for each pair of ports i, j on switch 2, *e.g.,* to compute a port-level traffic matrix.

Querying Paths. Full paths through a network may be described by combining atoms using the regular path combinators: concatenation (^), alternation (|), repetition (∗), intersection (&), and negation (∼). The most interesting combinator is concatenation: Given two path queries p1 and p2, the query p1 ^ p2 specifies a path that satisfies p1, takes a hop to the next switch, and then satisfies p2 from that point on. The interpretation of the other operators is natural: p1 | p2 specifies paths that satisfy *either* p1 *or* p2; p1* specifies paths that are zero or more repetitions of paths satisfying p1; p1 & p2 specifies paths that satisfy *both* p1 and p2, and ∼p1 specifies paths that do *not* satisfy p1.

Table 1 presents several useful queries that illustrate the utility of our system. Path queries enable novel capabilities (*e.g.,* localizing packet loss using just a few queries), significantly reduce operator labor (*e.g.,* measuring an accurate switch-level traffic matrix), and check policy invariants (*e.g.,* slice isolation) in the data plane.

Query Actions. An application can specify what to do with packets that match a query. For example, packets can be counted (*e.g.,* on switch counters), be sent out a specific port (*e.g.,* towards a collector), sent to the SDN controller, or extracted from sampling mechanisms (*e.g.,* sFlow). Below, we show Pyretic sample code for various use cases. Suppose that p is a path query defined according to the language (Fig. 3). Packets can be sent to abstract locations that "store" packets, called *buckets.* There are three types of buckets: *count buckets*, *packet buckets*, and *sampling buckets.* A count bucket is an abstraction that allows the application to count the packets going into it. Packets are not literally forwarded and held in controller data structures. In fact, the information content is stored in counters on switches. Below we illustrate the simplicity of the programming model.

```
cb = count_bucket() // create count bucket
cb.register(f)      // process counts by callback f
p.set_bucket(cb)    // direct packets matching p
...                 // into bucket cb
cb.pull_stats()     // get counters from switches
```

Packets can be sent to the controller, using the packet buckets and an equally straightforward programming idiom. Similarly, packets can also be *sampled* using technologies like NetFlow [1] or sFlow [3] on switches.

In general, an application can ask packets matching path queries to be processed by an arbitrary *NetKAT* policy, *i.e.*, any forwarding policy that is a mathematical

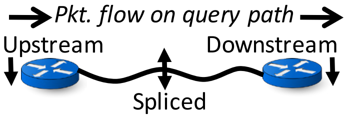


Figure 4: Query Capture Locations.

function from a packet to a set of packets [4,36]. The output packet set can be empty (*e.g.,* for dropped packets), or contain multiple packets (*e.g.,* for multicasted packets). For instance, packets matching a path query p can be forwarded out a specific mirroring port mp:

```
p.set_policy(fwd(mp)) // forward out mirror port
```

An arbitrarily complex Pyretic policy pol can be used instead of fwd above by writing p.set_policy(pol).

Query Capture Locations. The operator can specify where along a path to capture a packet that satisfies a query: either *downstream*—after it has traversed a queried trajectory, *upstream*—right as it enters the network, or *spliced*—somewhere in the middle. The difference between these three scenarios is illustrated in Fig. 4. The packets captured for the same query may differ at the three locations, because the network's forwarding policy may change as packets are in flight, or packets may be lost downstream due to congestion. For query p, the operator writes p.down() to ask matching packets to be captured downstream, p.up() to be captured upstream, p.updown() to be captured at both locations, and splice(p1,p2) to be captured between two sub-paths p1, p2 such that p = p1 ^ p2.

Sometimes, we wish to collect packets at many or even all points on a path rather than just one or two. The convenience function stitch(A,B,n) returns a set of queries by concatenating its first argument (*e.g.,* an in_atom) with *k* copies of its second argument (*e.g.,* an in_group), returning one query for each k in 0...n. For example, stitch(A,B,2) = {A, A^B, A^B^B}.

The capabilities described above allow the implementation of a *network-wide packet capture tool*. Drawing on wireshark terminology, an operator is now able to write global, *path-based capture filters* to collect exactly the packets matching a query.

4 Interactive Debugging with Path Queries

Consider a scenario shown in Fig. 5 where an operator is tasked with diagnosing a tenant's performance problem in a large compute cluster, where the connections between two groups of tenant virtual machines *A* and *B* suffer from poor performance with low throughput. The $A \to B$ traffic is routed along the four paths shown.

Such performance problems do occur in practice [75], yet are very challenging to diagnose, as none of the conventional techniques really help. Getting information from the end hosts' networking stack [62, 70] is difficult in virtualized environments. Coarse-grained packet sampling (NetFlow [1], [16]) may miss collecting the traffic relevant to diagnosis, *i.e.*, *A* and *B* traffic. Interface-level counters from the device may mask the problem entirely, as the issue occurs with just one portion of the traffic. It is possible to run wireshark [68] on switch CPUs; however this can easily impact switch performance and is very restrictive in its application [12]. Network operators may instead mirror a problematic subset of the traffic in the data plane through ACLs, *i.e.*, "match and mirror" [75]. However, this process is tedious and error-prone. The new monitoring rules must incorporate the results of packet modification in flight (*e.g.,* NATs and load balancers [45]), and touch several devices because of multi-path forwarding. The new rules must also be reconciled with overlapping existing rules to avoid disruption of regular packet forwarding. Ultimately, mirroring will incur large bandwidth and data collection overheads, corresponding to all mirrored traffic.

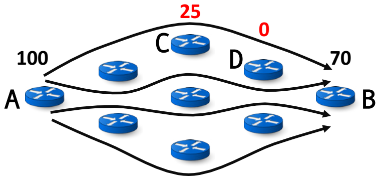


Figure 5: An example debugging scenario (§4).

In contrast, we show the ease with which a declarative query language and run-time system allow an operator to determine the root cause of the performance problem. In fact, the operator can perform *efficient* diagnosis using just switch counters—without mirroring any packets.

As a first step, the operator determines whether the end host or the network is problematic, by issuing a query counting all traffic that enters the network from *A* destined to *B*. She writes the query p1 below:

```
p1 = in_atom(srcip=vm_a, switch=s_a) ^ true*
    ^ out_atom(dstip=vm_b, switch=s_b)
p1.updown()
```

The run-time then provides statistics for $A \to B$ traffic, measured at network ingress (upstream) and egress (downstream) points. By comparing these two statistics, the operator can determine whether packets never left the host NIC, or were lost in the network.

Suppose the operator discovers a large loss rate in the network, as query p1 returns values 100 and 70 as shown in Fig. 5. Her next step is to localize the interface where most drops happen, using a downstream query p2:

```
probe_pred = switch=s_a & srcip=vm_a & dstip=vm_b
p2 = stitch(in_atom(probe_pred),
           in_group(true, ['switch']), 4)
```

These queries count $A \to B$ traffic on each switch-level path (and its prefix) from *A* to *B*. Suppose the runtime returns, among statistics for other paths, the packet counts 25 and 0 shown in red in Fig. 5. The operator concludes that link $C \rightarrow D$ along the first path has a high packet drop rate (all 25 packets dropped). Such packet drops may be due to persistent congestion or a faulty interface, affecting all traffic on the interface, or faulty rules in the switch (*e.g.,* a "silent blackhole" [75]) which affect just $A \rightarrow B$ traffic. To distinguish the two cases, the operator writes two queries measured midstream and downstream (each). Here are the midstream queries:

```
probe_pred = switch=s_a & srcip=vm_a & dstip=vm_b
p3 = splice((in_atom(probe_pred) ^ true*
             ^ in_atom(switch=s_c)),
            in_atom(switch=s_d))
p4 = splice(true* ^ in_atom(switch=s_c),
            in_atom(switch=s_d))
```

These queries determine the traffic loss rate on the $C \rightarrow D$ link, for all traffic traversing the link, as well as specifically the $A \rightarrow B$ traffic. By comparing these two loss rates, the operator can rule out certain root causes in favor of others. For example, if the loss rate for $A \rightarrow B$ traffic is particularly high relative to the overall loss rate, it means that that just the $A \rightarrow B$ traffic is silently dropped.

5 Path Query Compilation

Query compilation translates a collection of independently specified queries, along with the forwarding policy, into data-plane rules that recognize all packets traversing a path satisfying a query. These rules can be installed either on switches with single-stage [34] or multi-stage [8] match-action tables.[1] We describe downstream query compilation in §5.1-§5.3, and upstream compilation in §5.4. Downstream query compilation consists of three main stages:

1. We convert the regular expressions corresponding to the path queries into a DFA (§5.1).
2. Using the DFA as an intermediate representation, we generate state-transitioning (*i.e.*, *tagging*) and accepting (*i.e.*, *capture*) data-plane rules. These allow switches to match packets based on the state value, rewrite state, and capture packets which satisfy one or more queries (§5.2).
3. Finally, the run-time combines the query-related packet-processing actions with the regular forwarding actions specified by other controller applications. This is necessary because the state match and rewrite actions happen on the *same* packets that are forwarded by the switches (§5.3).

The run-time expands group atoms into the corresponding basic atoms by a pre-processing pass over the queries (we elide the details here). The resulting queries only contain `in`, `out`, and `in_out` atoms. We describe query compilation through the following simple queries:

```
p1 = in_atom(srcip=H1 & switch=1) ^
       out_atom(switch=2 & dstip=H2)
p2 = in_atom(switch=1) ^ in_out_atom(true, switch=2)
```

5.1 From Path Queries to DFAs

We first compile the regular path queries into an equivalent DFA,[2] in three steps as follows.

Rewriting atoms to in-out-atoms. The first step is quite straightforward. For instance, the path query `p1` is rewritten to the following:

```
in_out_atom(srcip=H1 & switch=1, true) ^
in_out_atom(true, switch=2 & dstip=H2)
```

Converting queries to regular expressions. In the second step, we convert the path queries into string regular expressions, by replacing each predicate by a character literal. However, this step is tricky: a key constraint is that different characters of the regular expressions cannot represent overlapping predicates (*i.e.*, predicates that can match the same packet). If they do, we may inadvertently generate an NFA (*i.e.*, a single packet might match two or more outgoing edges in the automaton). To ensure that characters represent non-overlapping predicates, we devise an algorithm that takes an input set of predicates *P*, and produces the smallest orthogonal set of predicates *S* that matches all packets matching *P*. The key intuition is as follows. For each new predicate `new_pred` in *P*, the algorithm iterates over the current predicates `pred` in *S*, teasing out new disjoint predicates and adding them to *S*:

```
int_pred = pred & new_pred
new_pred = new_pred & ~int_pred
pred = pred & ~int_pred
```

Finally, the predicates in *S* are each assigned a unique character. The full algorithm is described in Appendix B.

For the running example, Fig. 6 shows the emitted characters (for the partitioned predicates) and regular expressions (for input predicates not in the partitioned set). Notice in particular that the `true` predicate coming in to a switch is represented not as a single character but as an alternation of three characters. Likewise with `switch=1`, `switch=2`, and `true` (out). The final regular expressions for the queries `p1` and `p2` are:

```
p1: a^(c|e|g)^(a|d|f)^c
p2: (a|d)^(c|e|g)^(a|d|f)^(c|e)
```

[1] The compiler performs significantly better with multi-stage tables.

[2] We could conceivably use an NFA instead of a DFA, to produce fewer states. However, using an NFA would require each packet to store all the possible states that it might inhabit at a given time, and require switches to have a rule for each subset of states—leading to a large number of rules. Hence, we compile our path queries to a DFA.

Predicate	Regex	Predicate	Regex
`switch=1 & srcip=H1`	`a`	`∼switch=1`	`f`
`switch=1 & ∼srcip=H1`	`d`	`∼switch=2`	`g`
`switch=2 & dstip=H2`	`c`	`switch=1`	`a|d`
`switch=2 & ∼dstip=H2`	`e`	`switch=2`	`c|e`
`true` (in)	`a|d|f`	`true` (out)	`c|e|g`

Figure 6: Strings emitted for the running example (§5.1).

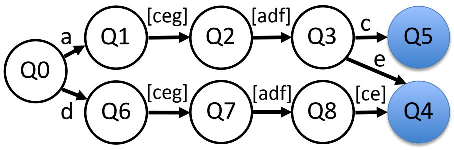


Figure 7: Automaton for `p1` and `p2` together. State `Q4` accepts `p2`, while `Q5` accepts both `p1` and `p2`.

Constructing the query DFA. Finally, we construct the DFA for `p1` and `p2` together using standard techniques. The DFA is shown in Fig. 7. For clarity, state transitions that reject packets from both queries are not shown.

5.2 From DFA to Tagging/Capture Rules

The next step is to emit policies that implement the DFA. Conceptually, we have two goals. First, for each packet, a switch must read the DFA state, identify the appropriate transition, and rewrite the DFA state. This action must be done once at switch ingress and egress. Second, if the packet's new DFA state satisfies one or more queries, we must perform the corresponding query actions, *e.g.,* increment packet or byte counts.

State transitioning policies. The high-level idea here is to construct a "test" corresponding to each DFA transition, and rewrite the packet DFA state to the destination of the transition if the packet passes the test. This is akin to a string-matching automaton checking if an input symbol matches an outgoing edge from a given state. To make this concrete, we show the intermediate steps of constructing the transitioning policy in Pyretic code.

We briefly introduce the notions of *parallel and sequential composition* of network policies, which we use to construct the transitioning policy. We treat each network policy as a mathematical function from a packet to a set of packets, similar to NetKAT and Pyretic [4, 36]. For example, a *match* `srcip=10.0.0.1` is a function that returns the singleton set of its input packet if the packet's source IP address is `10.0.0.1`, and an empty set otherwise. Similarly, a *modification* `port←2` is a function that changes the "port" field of its input packet to 2. Given two policies `f` and `g`—two functions on packets to sets of packets—the *parallel composition* of these two policies is defined as:

```
(f + g)(pkt) = f(pkt) ∪ g(pkt)
```

The *sequential composition* of policies is defined as:

Concept	Example	Description
Modification	`port←2`	Rewrites a packet field
Match	`switch=2`	Filters packets
Parallel composition	`monitor + route`	The union of results from two policies.
Sequential composition	`balance >> route`	Pipe the output from the first in to the second
Edge predicate	`pred_of(c)`	Get predicate of symbol
Path policy	`p.policy()`	Policy to process packets accepted by query `p`.

Figure 8: Syntactic Constructs in Query Compilation.

$$\texttt{(f >> g)(pkt)} = \cup_{pkt' \in f(pkt)} \texttt{g(pkt')}$$

For example, the policy

```
(srcip=10.0.0.1 + dstip=10.0.0.2) >> (port←2)
```

selects packets with either `srcip 10.0.0.1` or `dstip 10.0.0.2` and forwards them out of port 2 of a switch.

Now we produce a policy fragment for each edge of the DFA. Suppose the helper function `pred_of` takes in a character input `c` and produces the corresponding predicate. For each edge from state `s` to state `t` that reads character `c`, we construct the fragment

```
state=s & pred_of(c) >> state←t
```

We combine these fragments through parallel composition, which joins the tests and actions of multiple edges:

```
tagging = frag_1 + frag_2 + ... + frag_n
```

We produce *two* state transitioning policies, one each for ingress and egress actions. Each edge fragment belongs to exactly one of the two policies, and it is possible to know which one since we generate disjoint characters for these two sets of predicates. For example, here is part of the ingress transitioning policy for the DFA in Fig. 7:

```
in_tagging =
  state=Q0 & switch=1 & srcip=H1 >> state←Q2 +
  state=Q0 & switch=1 & ∼srcip=H1 >> state←Q6 +
   ... +
  state=Q7 & ∼switch=1 >> state←Q8
```

Accepting policies. The accepting policy is akin to the *accepting* action of a DFA: a packet that "reaches" an accepting state has traversed a path that satisfies some query; hence the packet must be processed by the actions requested by applications. We construct the accepting policy by combining edge fragments which move packets to accepting states. We construct the fragment

```
state=s & pred_of(c) >> p.policy()
```

for each DFA edge from state `s` to `t` through character `c`, where `t` is a state accepting query `p`. Here `p.policy()` produces the action that is applied to packets matching query `p`. Next we construct the *accepting* policy by a parallel composition of each such fragment:

```
capture = frag_1 + frag_2 + ... + frag_n
```

Similar to the transitioning policies, we construct two accepting policies corresponding to switch ingress and egress predicates. For example, for the DFA in Fig. 7, part of the accepting policy looks as follows:

```
out_capture =
  state=Q3 & switch=2 & dstip=H2 >> p1.policy()
  + ... +
  state=Q8 & switch=2 & dstip=H2 >> p2.policy()
```

Ingress tagging and Egress un-tagging. The run-time ensures that packets entering a network are tagged with the initial DFA state Q0. Symmetrically, packets leaving the network are stripped of their tags. We use the VLAN header to tag packets, but other mechanisms are possible.

5.3 Composing Queries and Forwarding

The run-time system needs to combine the packet-processing actions from the transitioning and accepting policies with the forwarding policy. However, this requires some thought, as all of these actions affect the *same* packets. Concretely, we require that:

1. packets are forwarded through the network normally, independent of the existence of queries,
2. packet tags are manipulated according to the DFA,
3. packets matching path queries are processed correctly by the application-programmed actions, and
4. no unnecessary duplicate packets are generated.

To achieve these goals, the run-time system combines the constituent policies as follows:

```
(in_tagging >> forwarding >> out_tagging)
+ (in_capture)
+ (in_tagging >> forwarding >> out_capture)
```

The first sequential composition (involving the two `tagging` policies and the `forwarding`) ensures both that forwarding continues normally (goal 1) as well as that DFA actions are carried out (goal 2). This works because `tagging` policies do not drop packets, and the `forwarding` does not modify the DFA state.[3] The remaining two parts of the top-level parallel composition (involving the two `capture` policies) ensure that packets reaching accepting states are processed by the corresponding query actions (goal 3). Finally, since each parallelly-composed fragment either forwards packets normally or captures it for the accepted query, no unnecessary extra packets are produced (goal 4).

Translating to match-action rules in switches. The run-time system hands off the composed policy above to Pyretic, which by default compiles it down to a single match-action table [20,36]. We also leverage *multi-stage tables* on modern switches [8, 42] to significantly improve compilation performance (§6). We can rewrite the joint policy above as follows:

```
(in_tagging + in_capture)
 >> forwarding
 >> (out_tagging + out_capture)
```

This construction preserves the semantics of the original policy provided `in_capture` policies do not forward packets onward through the data plane. This new representation decomposes the complex compositional policy into a *sequential pipeline* of three smaller policies—which can be independently compiled and installed to separate stages of match-action tables. Further, this enables *decoupling updates* to the query and forwarding rules on the data plane, allowing them to evolve independently at their own time scales.

5.4 Upstream Path Query Compilation

Upstream query compilation finds those packets at network ingress that *would* match a path, based on the current forwarding policy—assuming that packets are not dropped (due to congestion) or diverted (due to updates to the forwarding policy while the packets are in flight). We compile upstream queries in three steps, as follows.

Compiling using downstream algorithms. The first step is straightforward. We use algorithms described in sections §5.1-§5.3 to compile the set of upstream queries using *downstream* compilation. The output of this step is the *effective* forwarding policy of the network incorporating the behavior both of forwarding and queries. Note that we do *not* install the resulting rules on the switches.

Reachability testing for accepted packets. In the second step, we cast the upstream query compilation problem as a standard network *reachability test* [30, 31], which asks which of all possible packet headers at a source can reach a destination port with a specific set of headers. Such questions can be efficiently answered using *header space analysis* [30]: we simply ask which packets at network ingress, when forwarded by the *effective* policy above, *reach* header spaces corresponding to accepting states for query p. We call this packet match `upstream(p)`.

Capturing upstream. The final step is to process the resulting packet headers from reachability testing with application-specified actions for each query. We generate an *upstream capture* policy for queries `p1, ..., pn`:

```
(upstream(p1) >> p1.policy()) + ...
+ (upstream(pn) >> pn.policy())
```

We can implement complex applications of header space analysis like loop and slice leakage detection [30, §5] simply by compiling the corresponding upstream

[3] The run-time ensures this by constructing `tagging` policies with a virtual header field [36] that regular `forwarding` policies do not use.

path query [39]. Spliced queries can be compiled in a manner very similar to upstream queries.

In general, reachability testing does not restrict the paths taken to reach the destination—however, we are able to use the packet DFA state to do exactly that.

6 Optimizations

We implemented several key optimizations in our prototype to reduce query compile time and data-plane rule space. Later we show the quantitative impact of these optimizations (§7, Table 2). We briefly discuss the key ideas here; full details are in an extended version [39].

Cross-product explosion. We first describe the "cross-product explosion" problem that results in large compilation times and rule-sets when compiling the policies resulting from algorithms in §5. The output of NetKAT policy compilation is simply a prioritized list of match-action rules, which we call a *classifier.* When two classifiers C_1 and C_2 are composed—using parallel or sequential composition (§5.2, Fig. 8)—the compiler must consider the effect of every rule in C_1 on every rule in C_2. If the classifiers have N_1 and N_2 rules (resp.), this results in a $\Theta(N_1 \times N_2)$ operations. A similar problem arises when predicates are partitioned during DFA generation (§5.1). In the worst case, the number of orthogonal predicates may grow exponentially on the input predicate set, since every pair of predicates may possibly overlap.

Prior works have observed similar problems [15, 22, 58, 72]. Our optimizations reduce large compile time and rule sets through the domain-specific techniques below.

(A) Optimizing Conditional Policies. The policy generated from the state machine (§5.2) has a very special structure, namely one that looks like a conditional statement: *if state=s1 then ... else if state=s2 then ... else if ....* A natural way to compile this down is through the parallel composition of policies that look like `state=s_i >> state_policy_i`. This composition is expensive, because the classifiers of `state_policy_i` for all `i`, $\{C_i\}_i$, must be composed parallelly. We avoid computing these cross-product rule compositions as follows: If we ensure that each rule of C_i is specialized to match on packets disjoint from those of C_j—by matching on state `s_i`—then it is enough to simply *append* the classifiers C_i and C_j. This brings down the running time from $\Theta(N_i \times N_j)$ to $\Theta(N_i + N_j)$. We further compact each classifier C_i: we only add transitions to non-dead DFA states into `state_policy_i`, and instead add a default dead-state transition wherever a C_i rule drops the packets.

(B) Integrating tagging and capture policies. `Tagging` and `capture` policies have similar conditional structure:

```
tagging =                         capture =
  (cond1 >>  a1) +                  (cond1 >>  b1) +
  (cond2 >>  a2) +                  (cond2 >>  b2) +
  ...                                ...
```

Rather than supplying Pyretic with the policy `tagging + capture`, which will generate a large cross-product, we construct a simpler equivalent policy:

```
combined =
  (cond1 >>  (a1 + b1)) +
  (cond2 >>  (a2 + b2)) +
  ...
```

(C) Flow-space based pre-partitioning of predicates. In many queries, we observe that most input predicates are disjoint with each other, but predicate partitioning (§5.1) checks overlaps between them anyway. We avoid these checks by pre-partitioning the input predicates into disjoint flow spaces, and only running the partitioning within each flow space. For example, suppose in a network with `n` switches, we define `n` disjoint flow spaces `switch=1, ..., switch=n`. When a new predicate `pred` is added, we check if `pred & switch=i` is nonempty, and then *only* check overlaps with predicates intersecting the `switch=i` flow space.

(D) Caching predicate overlap decisions. We avoid redundant checks for predicate overlaps by caching the latest overlap results for all input predicates[4], and executing the remainder of the partitioning algorithm only when the cache is missed. Caching also enables introducing new queries incrementally into the network without recomputing all previous predicate overlaps.

(E) Decomposing query-matching into multiple stages. Often the input query predicates may have significant overlaps: for instance, one query may count on M source IP addresses, while another counts packets on N destination IP addresses. By installing these predicates on a single table stage, it is impossible to avoid using up $M \times N$ rules. However, modern switches [8, 43] support several match-action stages, which can be used to reduce rule space overheads. In our example, by installing the M source IP matches in one table and N destination matches in another, we can reduce the rule count to $M + N$. These smaller *logical* table stages may then be mapped to *physical* table stages on hardware [29, 56].

We devise an optimization problem to divide queries into groups that will be installed on different table stages. The key intuition is to spread queries matching on dissimilar header fields into multiple table stages to reduce rule count. We specify a cost function that estimates the worst-case rule space when combining predicates (Appendix A). The resulting optimization problem is NP-hard; however, we design a first-fit heuristic to group

[4] We index this cache by a hash on the string representation of the predicate's abstract syntax tree.

queries into table stages, given a limit on the number of stages and rule space per stage. The compilations of different stages are parallelizable.

(F) Detecting overlaps using Forwarding Decision Diagrams (FDDs). To make intersection between predicates efficient, we implement a recently introduced data structure called *Forwarding Decision Diagram* (FDD) [58]. An FDD is a binary decision tree in which each non-leaf node is a test on a packet header field, with two outgoing edges corresponding to true and false. Each path from the root to a leaf corresponds to a unique predicate which is the intersection of all tests along the path. Inserting a new predicate into the FDD only requires checking overlaps along the FDD paths *which the new predicate intersects,* speeding up predicate overlap detection.

7 Performance Evaluation

We evaluated the expressiveness of the query language and the debugging model in Table 1 and §4. Now, we evaluate the prototype performance quantitatively.

Implementation. We implemented the query language and compilation algorithms (§3, §5) on the Pyretic controller [36] and NetKAT compiler [58]. We extended the Hassel-C [48] implementation of header space analysis with inverse transfer function application for upstream compilation. NetFlow samples are processed with `nfdump` [41]. The query language is embedded in Python, and the run-time system is a library on top of Pyretic. The run-time sends switch rules to Open vSwitch [43] through OpenFlow 1.0 and the Nicira extensions [44]. We use `Ragel` [11] to compile string regular expressions. We evaluate our system using the `PyPy` compiler [50]. Our prototype is open-source [65].

Metrics. A path-query system should be efficient along the following dimensions:

1. Query compile time: Can a new query be processed at a “human debugging” time scale?
2. Rule set size: Can the emitted match-action rules fit into modern switches?
3. Tag set size: Can the number of distinct DFA states be encoded into existing tag fields?

There are other performance metrics which we do not report. Additional query rules that fit in the switch hardware tables do not adversely impact packet processing throughput or latency, because hardware is typically designed for deterministic forwarding performance.[5] The same principle applies to packet mirroring [47]. The time to install data plane rules varies widely depending on the switch used—prior literature reports between 1-20 milliseconds per flow setup [24]. Our compiler produces small rule sets that can be installed in a few seconds.

Experiment Setup. We pick a combination of queries from Table 1, including switch-to-switch traffic matrix, congested link diagnosis, DDoS source detection, counting packet loss per-hop per-path[6], slice isolation between two IP prefix slices, and firewall evasion. These queries involve broad scenarios such as resource accounting, network debugging, and enforcing security policy. We run our single-threaded prototype on an Intel Xeon E3 server with 3.70 GHz CPU (8 cores) and 32GB memory.

Compiling to a multi-stage table is much more efficient than single-stage table, since the former is not susceptible to cross-product explosion (§6). For example, the traffic matrix query incurs three orders of magnitude smaller rule space with the basic multi-stage setup (§5.3), relative to single-stage. Hence, we report multi-stage statistics throughout. Further, since optimization (E) decomposes queries into multiple stages (§6), and the stage compilations are parallelizable, we report the *maximum* compile time across stages whenever (E) is enabled.

(I) Benefit of Optimizations. We evaluate our system on an emulated Stanford campus topology [2], which contains 16 backbone routers, and over 100 network ingress interfaces. We measure the benefit of the optimizations when compiling *all of the queries listed above* together—collecting over 550 statistics from the network.[7]

The results are summarized in Table 2. Some trials did not finish[8], labeled “DNF.” Each finished trial shown is an average of five runs. The rows are keyed by optimizations—whose letter labels (A)-(F) are listed in paragraph headings in §6. We enable the optimizations one by one, and show the *cumulative* impact of all enabled optimizations in each row. The columns show statistics of interest—compile time (absolute value and factor reduction from the unoptimized case), *maximum* number of table rules (ingress and egress separately) on any network switch, and required packet DFA bits.

The cumulative compile-time reduction with all optimizations (last row) constitutes three orders of magnitude—reducing the compile time to about 5 seconds, suitable for interactive debugging by a human operator.[9][10] Further, the maximum number of rules required on any one switch fits comfortably in modern switch memory capacities [8, 14, 25]; and the DFA state

[5] Navindra Yadav. Personal communication, January 2016.

[6] We use the version of this query from §4, see `p2` there.

[7] By injecting traffic into the network, we tested that our system collects the right packets (Fig. 2), extracts the right switch counters, and produces no duplicate packets.

[8] The reason is that they run out of memory.

[9] Interactive response times within about 15 seconds retain a human in the “problem-solving frame of mind” [35, topic 11].

[10] We enable (F) only for larger networks, where the time to set up the data structure is offset by fast predicate intersection.

Enabled	Compile Time		Max # Rules		# State
Opts.	Abs. (s)	Rel. (X)	In	Out	Bits
None	> 4920	*baseline*	DNF	DNF	DNF
(A) only	> 4080	1.206	DNF	DNF	DNF
(A)-(B)	2991	1.646	2596	1722	10
(A)-(C)	56.19	87.48	1846	1711	10
(A)-(D)	35.13	139.5	1846	1711	10
(A)-(E)	5.467	894.7	260	389	16

Table 2: Benefit of optimizations on queries running on the Stanford campus topology. "DNF" means "Did Not Finish."

Network	# Nodes	Compile	Max # Rules		# State
		Time (s)	In	Out	Bits
Berkeley	25	10.7	58	160	6.0
Purdue	98	14.9	148	236	22.5
RF1755	87	6.6	150	194	16.8
RF3257	161	44.1	590	675	32.3
RF6461	138	21.4	343	419	29.2

Table 3: Performance on enterprise and ISP (L3) network topologies when all optimizations are enabled.

bits (2 bytes at most) fit within tag fields like VLANs. Finally, multi-stage query decomposition (E) reduces rule space usage significantly with more state bits.

(II) Performance on enterprise and ISP networks. We evaluate our prototype on real enterprise and inferred ISP networks, namely: UC Berkeley [6], Purdue University [63], and Rocketfuel (RF) topologies for ASes 1755, 3257 and 6461 [54, 61]. All optimizations are enabled. For each network, we report averages from 30 runs (five runs each of six queries). The results are summarized in Table 3. The average compile time is under 20 seconds in all cases but one; rule counts are within modern switch TCAM capacities; and DFA bits fit in an MPLS header.

(III) Scalability trends. We evaluate how performance scales with network size, on a mix of five synthetic ISP topologies generated from Waxman graphs [67] and IGen, a heuristic ISP topology generator [51]. We discuss the parameters used to generate the topologies in an extended version of this paper [39]. We report average metrics from 30 runs, *i.e.*, six queries compiled to five networks of each size. The trends are shown in Fig. 9. The average compile time (see red curve) is under ≈ 25 seconds until a network size of 140 nodes. In the same size range, the ingress table rule counts (see black curve) as well as the egress (not shown) are each under 700—which together can fit in modern switch TCAM memories. DFA packet bits (see numeric labels on black curve) fit in an MPLS header until 120 nodes.

For networks of about 140 nodes or smaller, our query system supports interactive debugging—continuing to provide useful results beyond for non-interactive tasks. We believe that these query compile times are a significant step forward for "human time scale" network debugging, which requires operators to be involved typically for hours [21, 75]. Among large ISP topologies mapped out in literature [61], each ISP can be supported in the "interactive" regime for PoP-level queries. We leave further scaling efforts, *e.g.,* to data centers, to future work.

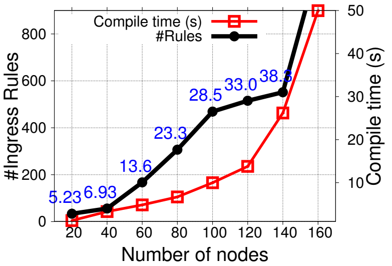


Figure 9: Scalability trends on synthetic ISP topologies. Numeric labels on the curve correspond to # DFA packet bits.

8 Related Work

We already discussed the most relevant prior works in §2; this section lays out other related work.

Data-plane query systems. Several query languages have been proposed for performing analytics over streams of packet data [7,13,20,66]. Unlike these works, we address the collection of *path-level* traffic flows, *i.e.*, observations of the same packets *across* space and time, which cannot be expressed concisely or achieved by (merely) asking for single-point packet observations.

Control-plane query systems. NetQuery [57] and other prior systems [9, 10, 26] allow operators to query information (*e.g.,* next hop for forwarding, attack fingerprints, *etc.*) from tuples stored on network nodes. As such, these works do not query the data plane. SIMON [40] and ndb [33] share our vision of interactive debugging, but focus on isolating control plane bugs.

Summary statistics monitoring systems. DREAM [37], ProgME [72] and OpenSketch [71] answer a different set of monitoring questions than our work, *e.g.,* detecting heavy hitters and changes in traffic patterns.

Programming traffic flow along paths. Several prior works [17, 27, 53, 60] aim to forward packets along policy-compliant paths. However, our work *measures* traffic along operator-specified paths, while the usual forwarding policy continues to handle traffic.

9 Conclusion

We have shown how to take a declarative specification for path-level measurements, and implement it in the data plane with accurate results at low overhead. We believe that this capability will be useful for network operators for better real-time problem diagnosis, security policy enforcement, and capacity planning.

Acknowledgments

This work was supported by NSF grants CCF-1535948 and CNS-1111520, and gifts from Intel, Huawei, and Cisco. We thank our shepherd, Oriana Riva, and the anonymous reviewers for their thoughtful feedback; Vimal Jeyakumar, Laurent Vanbever and Anirudh Sivaraman for helpful discussions; Nick Feamster, Vimal Jeyakumar, Divjyot Sethi, Nanxi Kang, Ronaldo Ferreira, Bharath Balasubramanian, Swarun Kumar, Xin Jin and Xuan (Kelvin) Zou for feedback on earlier drafts; Nate Foster and Arjun Guha for their generous help to integrate the NetKAT compiler with the Pyretic run-time; and Changhoon Kim and Arpit Gupta for suggestions to improve performance.

References

[1] Sampled Netflow, 2003. `http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html`.

[2] Mini-Stanford backbone topology, 2014. [Online, Retrieved February 17, 2016] `https://bitbucket.org/peymank/hassel-public/wiki/Mini-Stanford`.

[3] sFlow, 2015. `sflow.org`.

[4] Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. NetKAT: Semantic foundations for networks. In *Proc. ACM Symposium on Principles of Programming Languages* (2014).

[5] Andrew Lerner. The cost of downtime, 2014. [Online, Retrieved February 17, 2016] `http://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/`.

[6] Berkeley Information Services and Technology. UCB network topology. [Online, Retrieved February 17, 2016] `http://www.net.berkeley.edu/netinfo/newmaps/campus-topology.pdf`.

[7] Borders, K., Springer, J., and Burnside, M. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security* (2012).

[8] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proc. ACM SIGCOMM* (2013).

[9] Chen, X., Mao, Y., Mao, Z. M., and Van der Merwe, J. Decor: Declarative network management and operation. *ACM SIGCOMM Computer Communication Review* (2010).

[10] Cohen, J., Repantis, T., McDermott, S., Smith, S., and Wein, J. Keeping track of 70,000+ servers: The Akamai query system. In *Proc. Large Installation System Administration Conference, LISA* (2010).

[11] Colm Networks. Ragel state machine compiler. [Online, Retrieved February 17, 2016] `http://www.colm.net/open-source/ragel/`.

[12] Configuring Wireshark on the Catalyst 3850 switch. [Online, Retrieved January 25, 2016] `http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst3850/software/release/3se/network_management/configuration_guide/b_nm_3se_3850_cg/b_nm_3se_3850_cg_chapter_01000.html#concept_8EBE76A3D6DB46B79E7F0B6CFFE9FDF9`.

[13] Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V. Gigascope: A stream database for network applications. In *Proc. ACM SIGMOD* (2003).

[14] Curtis, A. R., Mogul, J. C., Tourrilhes, J., Yalagandula, P., Sharma, P., and Banerjee, S. DevoFlow: Scaling flow management for high-performance networks. In *Proc. ACM SIGCOMM* (2011).

[15] D'Antoni, L., and Veanes, M. Minimization of symbolic automata. In *Proc. ACM Symposium on Principles of Programming Languages* (2014).

[16] Duffield, N. G., and Grossglauser, M. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Networking* (June 2001).

[17] Fayazbakhsh, S. K., Sekar, V., Yu, M., and Mogul, J. C. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2014).

[18] Feldmann, A., Greenberg, A., Lund, C., Reingold, N., and Rexford, J. NetScope: Traffic engineering for IP networks. *IEEE Network* (2000).

[19] Feldmann, A., Greenberg, A., Lund, C., Reingold, N., Rexford, J., and True, F. Deriving traffic demands for operational IP networks: Methodology and experience. *IEEE/ACM Trans. Networking* (June 2001).

[20] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. Frenetic: A network programming language. In *Proc. ACM International Conference on Functional Programming* (2011).

[21] Gill, P., Jain, N., and Nagappan, N. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proc. ACM SIGCOMM* (2011).

[22] Gupta, A., Vanbever, L., Shahbaz, M., Donovan, S. P., Schlinker, B., Feamster, N., Rexford, J., Shenker, S., Clark, R., and Katz-Bassett, E. SDX: A software defined Internet exchange. In *Proc. ACM SIGCOMM* (2014).

[23] Handigol, N., Heller, B., Jeyakumar, V., Maziéres, D., and McKeown, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2014).

[24] He, K., Khalid, J., Das, S., Gember-Jacobson, A., Prakash, C., Akella, A., Li, L. E., and Thottan, M. Latency in software defined networks: Measurements and mitigation techniques. In *Proc. ACM SIGMETRICS* (2015).

[25] Huang, D. Y., Yocum, K., and Snoeren, A. C. High-fidelity switch models for software-defined network emulation. In *Proc. Hot Topics in Software Defined Networks* (2013).

[26] Huebsch, R., Hellerstein, J. M., Lanham, N., Loo, B. T., Shenker, S., and Stoica, I. Querying the Internet with PIER. In *Proc. International Conference on Very Large Data Bases* (2003).

[27] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., and Vahdat, A. B4: Experience with a globally-deployed software defined WAN. In *Proc. ACM SIGCOMM* (2013).

[28] Jeyakumar, V., Alizadeh, M., Geng, Y., Kim, C., and Mazières, D. Millions of little minions: Using packets for low latency network programming and visibility. In *Proc. ACM SIGCOMM* (2014).

[29] Jose, L., Yan, L., Varghese, G., and McKeown, N. Compiling packet programs to reconfigurable switches. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2015).

[30] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2012).

[31] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying network-wide invariants in real time. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2013).

[32] Kim, C., Sivaraman, A., Katta, N., Bas, A., Dixit, A., and Wobker, L. J. In-band network telemetry via programmable dataplanes, June 2015. Demo at Symposium on SDN Research, `http://opennetsummit.org/wp-content/themes/ONS/files/sosr-demos/sosr-demos15-final17.pdf`.

[33] Lin, C.-C., Caesar, M., and van der Merwe, K. Toward interactive debugging for ISP networks. In *Proc. ACM Workshop on Hot Topics in Networking* (2009).

[34] 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* (2008).

[35] Miller, R. B. Response time in man-computer conversational transactions. In *Proc. Fall Joint Computer Conference, Part I* (1968).

[36] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing software-defined networks. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2013).

[37] Moshref, M., Yu, M., Govindan, R., and Vahdat, A. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proc. ACM SIGCOMM* (2014).

[38] Narayana, S., Rexford, J., and Walker, D. Compiling path queries in software-defined networks. In *Proc. Hot Topics in Software Defined Networks* (2014).

[39] Narayana, S., Tahmasbi, M., Rexford, J., and Walker, D. Compiling Path Queries (Extended version). Tech. rep., Princeton University, 2015. [Online, Retrieved February 17, 2016] `http://www.cs.princeton.edu/~narayana/pathqueries`.

[40] Nelson, Tim and Yu, Da and Li, Yiming and Fonseca, Rodrigo and Krishnamurthi, Shriram. Simon: Scriptable interactive monitoring for SDNs. In *Proc. ACM Symposium on SDN Research* (2015).

[41] NFDUMP tool suite. [Online, Retrieved February 17, 2016] `http://nfdump.sourceforge.net/`.

[42] Openflow v1.3 Specification. [Online, Retrieved February 17, 2016] `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf`.

[43] OpenVSwitch. [Online, Retrieved August 15, 2015] `openvswitch.org`.

[44] OpenVSwitch Nicira Extensions. [Online, Retrieved February 17, 2016] `http://git.openvswitch.org/cgi-bin/gitweb.cgi?p=openvswitch;a=blob;f=include/openflow/nicira-ext.h`.

[45] Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D. A., Kern, R., Kumar, H., Zikos, M., Wu, H., Kim, C., and Karri, N. Ananta: Cloud scale load balancing. In *Proc. ACM SIGCOMM* (2013).

[46] Path Queries for Interactive Network Debugging. [Online, Retrieved February 17, 2016] `http://www.cs.princeton.edu/~narayana/pathqueries`.

[47] Performance impact of SPAN on the different Catalyst platforms. [Online, Retrieved January 21, 2016] `http://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/10570-41.html#anc48`.

[48] Peyman Kazemian. Hassel: Header space library. [Online, Retrieved February 17, 2016] `https://bitbucket.org/peymank/hassel-public/wiki/Home`.

[49] Postel, J. IP record route (Internet Protocol), 1981. RFC 791 [Online, Retrieved February 17, 2016] `http://www.ietf.org/rfc/rfc791.txt`.

[50] PyPy. [Online, Retrieved February 17, 2016] `http://pypy.org`.

[51] Quoitin, B., Van den Schrieck, V., FranÃğois, P., and Bonaventure, O. IGen: Generation of router-level Internet topologies through network design heuristics. In *Proc. International Teletraffic Congress* (2009).

[52] Rasley, J., Stephens, B., Dixon, C., Rozner, E., Felter, W., Agarwal, K., Carter, J., and Fonseca, R. Planck: Millisecond-scale monitoring and control for commodity networks. In *Proc. ACM SIGCOMM* (2014).

[53] Reitblatt, M., Canini, M., Guha, A., and Foster, N. FatTire: Declarative fault tolerance for software-defined networks. In *Proc. Hot Topics in Software Defined Networks* (2013).

[54] Rocketfuel: An ISP topology mapping engine. [Online, Retrieved February 17, 2016] `http://research.cs.washington.edu/networking/rocketfuel/interactive/`.

[55] Savage, S., Wetherall, D., Karlin, A., and Anderson, T. Practical network support for IP traceback. In *Proc. ACM SIGCOMM* (2000).

[56] Schlesinger, C., Greenberg, M., and Walker, D. Concurrent NetCore: From policies to pipelines. In *Proc. ACM International Conference on Functional Programming* (2014).

[57] Shieh, A., Sirer, E. G., and Schneider, F. B. NetQuery: A knowledge plane for reasoning about network properties. In *Proc. ACM SIGCOMM* (2011).

[58] Smolka, S., Eliopoulos, S., Foster, N., and Guha, A. A fast compiler for NetKAT. In *Proc. ACM International Conference on Functional Programming* (2015).

[59] Snoeren, A. C., Partridge, C., Sanchez, L. A., Jones, C. E., Tchakountio, F., Kent, S. T., and Strayer, W. T. Hash-based IP traceback. In *Proc. ACM SIGCOMM* (2001).

[60] Soulé, R., Basu, S., Marandi, P. J., Pedone, F., Kleinberg, R., Sirer, E. G., and Foster, N. Merlin: A language for provisioning network resources. In *Proc. ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (Dec. 2014).

[61] Spring, N., Mahajan, R., and Wetherall, D. Measuring ISP topologies with Rocketfuel. In *Proc. ACM SIGCOMM* (2002).

[62] Sun, P., Yu, M., Freedman, M. J., and Rexford, J. Identifying performance bottlenecks in CDNs through TCP-level monitoring. In *Proc. of the First ACM SIGCOMM Workshop on Measurements Up the Stack* (2011).

[63] Sung, Y.-W. E., Rao, S. G., Xie, G. G., and Maltz, D. A. Towards systematic design of enterprise networks. In *Proc. ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2008).

[64] TAMMANA, P., AGARWAL, R., AND LEE, M. Cherrypick: Tracing packet trajectory in software-defined datacenter networks. In *Proc. ACM Symposium on SDN Research* (2015).

[65] THE PYRETIC LANGUAGE AND RUN-TIME SYSTEM. [Online, Retrieved February 17, 2016] `https://github.com/frenetic-lang/pyretic`.

[66] UDDIN, M. Real-time search in large networks and clouds, 2013.

[67] WAXMAN, B. Routing of multipoint connections. *IEEE J. on Selected Areas in Communications* (1988).

[68] WIRESHARK. [Online, Retrieved February 17, 2016] `https://www.wireshark.org`.

[69] WU, W., WANG, G., AKELLA, A., AND SHAIKH, A. Virtual network diagnosis as a service. In *Proc. Symposium of Cloud Computing* (2013).

[70] YU, M., GREENBERG, A. G., MALTZ, D. A., REXFORD, J., YUAN, L., KANDULA, S., AND KIM, C. Profiling network performance for multi-tier data center applications. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2011).

[71] YU, M., JOSE, L., AND MIAO, R. Software defined traffic measurement with OpenSketch. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2013).

[72] YUAN, L., CHUAH, C.-N., AND MOHAPATRA, P. ProgME: Towards programmable network measurement. In *Proc. ACM SIGCOMM* (2007).

[73] ZHANG, H., LUMEZANU, C., RHEE, J., ARORA, N., XU, Q., AND JIANG, G. Enabling layer 2 pathlet tracing through context encoding in software-defined networking. In *Proc. Hot Topics in Software Defined Networks* (2014).

[74] ZHANG, HARVEST AND REICH, JOSHUA AND REXFORD, JENNIFER. Packet traceback for software-defined networks. Tech. rep., Princeton University, 2015. [Online, Retrieved September 10, 2015] `https://www.cs.princeton.edu/research/techreps/TR-978-15`.

[75] ZHU, Y., KANG, N., CAO, J., GREENBERG, A., LU, G., MAHAJAN, R., MALTZ, D., YUAN, L., ZHANG, M., ZHAO, B., AND ZHENG, H. Packet-level telemetry in large data-center networks. In *Proc. ACM SIGCOMM* (2015).

A Multi-stage rule-packing problem

Below we write down the integer optimization problem that minimizes the number of table stages subject to constraints on the number of stages and rule space available per stage. Typically, predicate partitioning time is proportional to the size of the output set of predicates, so this also reduces the compile time significantly:

```
minimize: S = $\sum_j y_j$
variables: $q_{ij} \in \{0,1\}, y_j \in \{0,1\}$
subject to:
  $\forall j:$ $\text{cost}(\{q_{ij} : q_{ij} = 1\}) \leq \text{rulelimit} * y_j$
  $\forall i:$ $\sum_j q_{ij} = 1$
  $S \leq \text{stagelimit}$
```

Here the variable q_{ij} is assigned a value 1 if query `i` is assigned to stage `j`, and 0 otherwise. The variable y_j is 1 if stage `j` is used by at least one query and 0 otherwise. The constraints ensure, respectively, that (i) queries in a given stage respect the rule space limits for that stage, (ii) every query is assigned exactly one table stage, and that (iii) the total number of stages is within the number of maximum stages supported by the switch. The optimization problem minimizes the number of used table stages, which is a measure of the latency and complexity of the packet-processing pipeline.

We now write down the cost function that determines the rule space usage of a bunch of queries together. First, we define the *type* and *count* for each query as the set of header fields the query matches on, and the number of total matches respectively. In the example in §6, the query types and counts would be `q1: (['srcip'], 100), q2: (['dstip'], 200), q3: (['srcip'], 300)`. We estimate the *worst-case* rule space cost[11] of putting two queries together into one stage as follows:

```
cost ((type1, count1), (type2, count2)) :=
  case type1 == φ:
    count2 + 1
  case type1 == type2:
    count1 + count2
  case type1 ⊂ type2:
    count1 + count2
  case type1 ∩ type2 == φ:
    (count1 + 1) * (count2 + 1) - 1
  case default:
    (count1 + 1) * (count2 + 1) - 1
```

The type of the resulting query is `type1 ∪ type2`, as the predicate partitioning (Alg. 1) creates matches with headers involving the union of the match fields in the two queries. Hence, we can construct a function which produces a new query type and count, given two existing query types and counts. It is easy to generalize this function to more than two arguments by iteratively applying it to the result of the previous function application and the next query[12]. Hence, we can compute the worst-case rule space cost of putting a bunch of queries together into one stage.

Our cost function and formulation are different from prior works that map logical to physical switch tables [29, 56] for two reasons. First, query predicates can be installed on any table: there is no ordering or dependency between them, so there are more possibilities to explore in our formulation. Second, our rule space cost function explicitly favors predicates with similar headers in one table, while penalizing predicates with very different header matches.

Reduction of bin-packing to rule-packing. It is straightforward to show that the problem of minimizing

[11] It is in general difficult to compute the exact rule space cost of installing two queries together in one stage without actually doing the entire overlap computation in Alg. 1.

[12] We believe, but are yet to show formally, that this cost function is associative.

Algorithm 1 Predicate partitioning (§5.1).

```
 1: $P = set\_of\_predicates$
 2: $S = \emptyset$
 3: for $new\_pred \in P$ do
 4:     for $pred \in S$ do
 5:         if $pred$ is equal to $new\_pred$ then
 6:             continue the outer loop
 7:         else if $pred$ is a superset of $new\_pred$ then
 8:             $difference = pred \ \& \sim new\_pred$
 9:             $S \leftarrow S \cup \{difference, new\_pred\}$
10:             $S \leftarrow S \setminus \{pred\}$
11:             continue the outer loop
12:         else if $pred$ is a subset of $new\_pred$ then
13:             $new\_pred \leftarrow new\_pred \ \& \sim pred$
14:         else if intersect then
15:             $inter_1 = pred \ \& \sim new\_pred$
16:             $inter_2 = \sim pred \ \& \ new\_pred$
17:             $inter_3 = pred \ \& \ new\_pred$
18:             $S \leftarrow S \cup \{inter_1, inter_2, inter_3\}$
19:             $S \leftarrow S \setminus \{pred\}$
20:             $new\_pred \leftarrow new\_pred \ \& \sim pred$
21:         end if
22:     end for
23:     $S \leftarrow S \cup \{new\_pred\}$
24: end for
```

the number of bins B of capacity V while packing n items of sizes $a_1, a_2, \cdots, a_n$ can be solved through a specific instance of the rule packing problem above. We construct n queries of the same type, with rule counts $a_1, \cdots, a_n$ respectively. We set the `rulelimit` to the size of the bins V, and `stagelimit` to the number of maximum bins allowed in the bin packing problem (typically n). Since all queries are of the same type, the rule space cost function is just the sum of the rule counts of the queries at a given stage. It is then easy to see that the original bin-packing problem is solved by this instance of the rule-packing problem.

First-fit Heuristic. The first-fit heuristic we use is directly derived from the corresponding heuristic for bin-packing. We fit a query into the first stage that allows the worst-case rule space blowup to stay within the prespecified per-stage `rulelimit`. The cost function above is used to compute the final rule-space after including a new query in a stage. We use a maximum of 10 logical stages in our experiments, with a 2000 rule limit per stage in the worst-case.

The logical stages match and modify completely disjoint parts of the packet state. We believe that a packet program compiler, *e.g.,* [29], can efficiently lay out the query rules on a physical switch table, since there are no dependencies between these table stages.

B Predicate Partitioning

To ensure that characters represent non-overlapping predicates, we apply Alg. 1 to partition the input predicates. The algorithm takes an input set of predicates `P`, and produces an orthogonal set of predicates `S`.

The partitioned set `S` is initialized to a null set (line 2). We iterate over the predicates in `P`, teasing out overlaps with existing predicates in `S`. If the current input predicate `new_pred` already exists in `S`, we move on to the next input (lines 5-6). If a predicate `pred` in `S` is a superset of `new_pred`, we split `pred` into two parts, corresponding to the parts that do and don't overlap with `new_pred` (lines 7-11). Then we move to the next input predicate. The procedure is symmetrically applied when `pred` is a subset of `new_pred` (lines 12-13), except that we continue looking for more predicates in `S` that may overlap with `new_pred`. Finally, if `pred` and `new_pred` merely intersect (but neither is a superset of the other), we create three different predicates in `S` according to three different combinations of overlap between the two predicates (lines 14-20). Finally, any remaining pieces of `new_pred` are added to the partitioned set `S`. Under each case above and for each predicate in `P`, we also keep track of the predicates in the partitioned set `S` with which it overlaps (details elided).

Simplifying Software-Defined Network Optimization Using SOL

Victor Heorhiadi, Michael K. Reiter, Vyas Sekar
UNC Chapel Hill, UNC Chapel Hill, Carnegie Mellon University

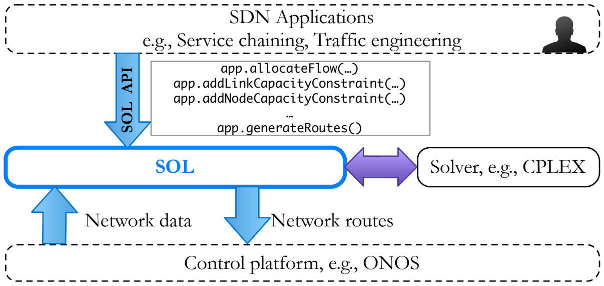
Abstract

Realizing the benefits of SDN for many network management applications (e.g., traffic engineering, service chaining, topology reconfiguration) involves addressing complex optimizations that are central to these problems. Unfortunately, such optimization problems require (a) significant manual effort and expertise to express and (b) non-trivial computation and/or carefully crafted heuristics to solve. Our goal is to simplify the deployment of SDN applications using *general* high-level abstractions for capturing optimization requirements from which we can *efficiently* generate optimal solutions. To this end, we present SOL, a framework that demonstrates that it is possible to simultaneously achieve generality and efficiency. The insight underlying SOL is that many SDN applications can be recast within a unifying *path-based* optimization abstraction. Using this, SOL can efficiently generate near-optimal solutions and device configurations to implement them. We show that SOL provides comparable or better scalability than custom optimization solutions for diverse applications, allows a balancing of optimality and route churn per reconfiguration, and interfaces with modern SDN controllers.

1 Introduction

Software-defined networking (SDN) is an enabler for network management applications that may otherwise be difficult to realize using existing control-plane mechanisms. Recent work has used SDN-based mechanisms to implement network configuration for a range of management tasks: traffic engineering (e.g., [40]), service chaining (e.g., [39]), energy efficiency (e.g., [19]), network functions virtualization (NFV) (e.g., [14]), and cloud offloading (e.g., [44]), among others.

While this body of work has been instrumental in demonstrating the potential benefits of SDN, realizing these benefits requires significant effort. In particular, at the core of many SDN applications are custom optimization problems to tackle various constraints and requirements that arise in practice (§2). For instance, an SDN application might need to account for limited TCAM, link capacities, or middlebox capacities, among other considerations. Developing such formulations involves a non-trivial learning curve, a careful understanding of theoretical and practical issues, and considerable manual effort. Furthermore, when the resulting optimization problems are intractable to solve with state-of-the-art solvers (e.g., CPLEX or Gurobi), heuristic algorithms must be crafted to ensure that new configurations can be generated on timescales demanded by the application as relevant inputs (e.g., traffic matrix entries) change (e.g., [19, 29]). Furthermore, without a common framework for representing network optimization tasks, it is difficult to reuse key ideas across applications or to combine useful features into a custom new application.


SDN Applications
e.g., Service chaining, Traffic engineering
SOL API
SOL
Solver, e.g., CPLEX
Network data
Network routes
Control platform, e.g., ONOS

```
app.allocateFlow(…)
app.addLinkCapacityConstraint(…)
app.addNodeCapacityConstraint(…)
…
app.generateRoutes()
```

Figure 1: ***Developers use the SOL high-level APIs to specify optimization goals and constraints. SOL generates near-optimal solutions and produces device configurations that are input to the SDN control platform.***

Our goal in this work is to raise the level of abstraction for writing such SDN-based network optimization applications. To this end, we introduce SOL, a framework that enables SDN application developers to express high-level application goals and constraints. Conceptually, SOL is an intermediate layer that sits between the SDN optimization applications and the actual control platform (see Fig. 1). Application developers who want to develop new network optimization capabilities express their requirements using the SOL API. SOL then generates configurations that meet these goals, which can be deployed to SDN control platforms.

There are two natural requirements for such a framework: (1) **generality** to express the requirements for a broad spectrum of SDN applications (e.g., traffic engineering, policy steering, load balancing, and topology management); and (2) **efficiency** to generate (near-) optimal configurations on a timescale that is responsive to application needs. Given the diversity of the application requirements and the trajectory of prior work in developing custom solutions (e.g., [39, 24, 23, 19, 29, 14, 8, 46, 40, 20]), generality and efficiency appear individually difficult, let alone combined. We show that it is indeed possible to achieve both generality and efficiency.

The key insight in SOL to achieve generality is that many network optimization problems can be expressed as *path-based* formulations. Paths are a natural abstraction for application developers to reason about intended network behaviors and to express policy requirements. For example, we can use paths to specify service chaining requirements (e.g., each path includes a firewall and intrusion-detection system, in that order) or redundancy (e.g., each includes two intrusion-prevention systems, in case one fails open). Finally, it is easy to model device (e.g., TCAM space, middlebox CPU) and link resource consumption based on the volume of traffic flowing through paths that traverse that device or link.

The natural question is whether the generality of path-based formulations precludes efficiency. Indeed, if implemented naively, optimization problems expressed over the paths that traffic might travel will introduce efficiency challenges since the number of paths grows exponentially with the network size. Our key insight is that by combining infrequent, offline preprocessing with simple, online *path-selection algorithms* (e.g., shortest paths or random paths), we can achieve near-optimal solutions in practice for all applications we considered. Moreover, SOL is typically far more efficient than solving the optimization problems originally used to express these applications' requirements.

We have implemented SOL as a Python-based library that interfaces with `ONOS` [5] and `OpenDaylight` [35] (§7). We have also prototyped numerous SDN applications in SOL, including SIMPLE [39], ElasticTree [19], Panopticon [29], and others of our own design (§6 and App. B). SOL is open-source; we have released modules for various applications coded in SOL as well as our `ONOS` extensions [22].

Our evaluations on a range of topologies show that: 1) SOL outperforms several applications' original optimization algorithms by an order of magnitude or more, and is even competitive with their custom heuristics; 2) SOL scales better than other network management tools like Merlin [45]; 3) SOL substantially reduces the effort required (e.g., in terms of lines of code) for implementing new SDN applications by an order of magnitude; and 4) optional SOL extensions can reduce route churn substantially across reconfigurations with modest impact on optimality.

2 Background and Motivation

In this section, we describe representative network applications that could benefit from a framework such as SOL. We highlight the need for careful formulation and algorithm development involved in prior efforts, as well as the diversity of requirements they entail.

2.1 Traffic engineering

Traffic engineering (TE) is a canonical application that was an early driving application for SDN [24, 23]. Fig. 2 shows an example where traffic classes C1 and C2 need to be routed completely while minimizing the load on the most heavily loaded link. A TE application takes as input traffic demands (e.g., the traffic matrix between WAN sites), a specification of the traffic classes and priorities, and the network topology and link capacities. It determines how to route each class to achieve network-wide objectives, e.g., minimizing congestion [11] or weighted max-min fairness [24, 23].

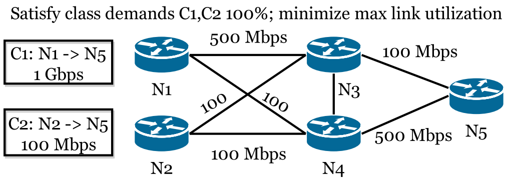


Figure 2: ***Traffic engineering applications***

Challenges: Simple goals like link congestion can be represented and solved via max-flow formulations [1]. However, the expressivity and efficiency quickly breaks down for more complex objectives such as max-min fairness, which multiple research efforts have sought to address [23, 9, 24]. When max-flow like formulations fail, designers invariably revert to "low-level" techniques such as linear programs (LP) or combinatorial algorithms. Neither is ideal—using/tuning LP solvers is painful as they expose a very low-level interface, and combinatorial algorithms require significant theoretical expertise. Finally, translating the algorithm output into actual routing rules requires care to install volume-aware rules to truly reap the benefits of the optimization [47].

2.2 Service chaining

Networks today rely on a wide variety of middleboxes (e.g., IDS, proxy, firewall) for performance, security, and external compliance capabilities (e.g., [44]). The goal of service chaining is to ensure that each class of traffic is routed through the desired sequence of network functions. For example, in Fig. 3, class C1 is required to traverse a firewall and proxy in order. Such policy routing rules must be suitably encoded within the available TCAM on SDN switches [39]. Since middleboxes are often compute-intensive, they can get easily overloaded and thus operators would like to balance the load on these appliances [39, 15]. The key inputs to such applications are the service chaining requirements of different classes, traffic demands, and the available middlebox processing resources. The application then sets up the forwarding rules such that the service chaining requirements are met while respecting the switch TCAM and middlebox capacities. Furthermore, as many of these middleboxes are stateful, these rules must ensure flow affinity.

Challenges: Service chaining introduces more complex requirements when compared to TE applications.

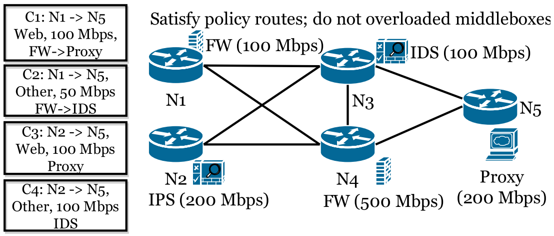


Figure 3: ***Service chaining applications***

First, modeling the consumption of switch TCAM introduces discrete components into the optimization, which impacts scalability [39]. Second, such service processing requirements fall outside the scope of existing network flow abstractions [8]. Third, service chaining highlights the complexity of combining different requirements; e.g., reasoning about the interaction between the load balancing algorithm and the switch TCAM constraints is non-trivial [25]. Existing service chaining efforts developed custom heuristics [7] or new theoretical extensions [8]. Furthermore, as observed previously, ensuring flow affinity can be quite tricky [21, 20].

2.3 Flexible topology management

SDN enables topology modifications that would be difficult to implement with existing control plane capabilities. For instance, ElasticTree [19] and Response [46] use SDN to dynamically switch on/off network links and nodes to make datacenters more energy efficient. In Fig. 4, these applications might shut down node N3 during periods of low utilization, if classes C1 and C2 can be routed via N4 without significantly impacting end-to-end performance. Topology reconfiguration is especially feasible in rich topologies with multiple paths between every source and destination. Such applications take as input the demand matrix (similar to the TE task) and then compute the nodes and links that should be active and traffic-engineered routes to ensure performance SLAs.

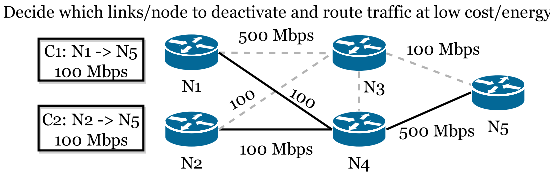


Figure 4: ***Topology reconfiguration applications***

Challenges: The on-off requirement on the switches/links once again introduces discrete constraints, yielding integer-linear optimizations that are theoretically intractable and difficult to express using max-flow like abstractions. Solving such problems requires significant computation even on small topologies and thus forces developers to design new, heuristic solving strategies; e.g., ElasticTree uses a greedy bin-packing algorithm [19].

2.4 Network function virtualization

Prior work has leveraged SDN capabilities to offload or outsource network functions to leverage clusters or clouds [44, 16, 40]. This is especially useful for expensive deep-packet-inspection services [20]. The key decision here is to decide how much of the processing on each path to offload to the remote datacenter — e.g., in Fig. 5, how much of class C1 traffic should be routed to the datacenter between N4 and N5 for IPS processing, versus processing it at N3. Offloading can increase user-perceived latency and impose additional load on network links. Moreover, some active functions (e.g., WAN optimizers or IPS) induce changes to the observed traffic volumes due to their actions. Thus, optimizing such offloading must take into account the congestion that might be introduced, as well as latency impact and any traffic volume changes induced by such outsourced functions. Further generalizations have considered not only offloading middlebox services but also elastically scaling them [36, 14, 34, 6], exacerbating these issues.

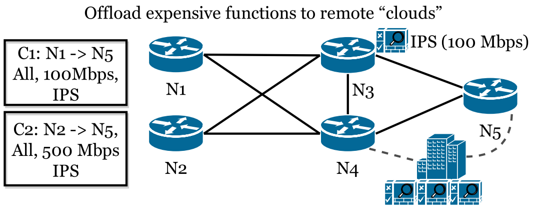


Figure 5: ***Offloading network functions***

Challenges: Such offloading and elastic scaling opportunities introduce new dimensions to optimization that are difficult to capture. For instance, offloading requires rerouting the traffic and thus optimizations must model the impact on link loads, downstream nodes, and TE objectives. If done naively, this can introduce non-linear dependencies since the actions of downstream nodes depend on control decisions made upstream. The active changes to traffic volumes by some functions (e.g., compression for redundancy elimination or drops by IPS) also introduce non-linear dependencies in the optimization. Finally, elastic scaling introduces a discrete aspect to the problem similar to the topology modification application, further decaying the problem's tractability.

2.5 Motivation for SOL

Drawing on the above discussion (and our own experience), we summarize a few key considerations:

- Network applications have diverse and complex optimization requirements; e.g., service chaining requires us to reason about valid paths while topology modification needs to enable/disable nodes.
- Designers of these applications have to spend significant effort in expressing and debugging these prob-

lems using low-level optimization libraries.

- It can take non-trivial expertise to ensure that the problems can be solved fast enough to be relevant for operational timescales, e.g., recomputing TE every few minutes or periodically solving the large integer-linear programs (ILPs) supporting topology reconfiguration (e.g., [19]).

3 SOL Overview

Our overarching vision in developing SOL is to raise the level of abstraction in developing new SDN applications and specifically to eliminate some of the black art in developing SDN-based optimizations, making them more accessible for deployment by network managers. To do so, SOL abstracts away low-level details of optimization solvers and SDN controllers, allowing the developer to focus on the high-level application goals (recall Fig. 1). SOL takes as inputs the network topology, traffic patterns, and optimization requirements in the SOL API. It then translates these into constraints for optimization solvers such as `CPLEX` or `Gurobi`. Finally, SOL interfaces with existing SDN control platforms such as `ONOS` to install the forwarding rules on the SDN switches. SOL does not require modifications to the existing control or data plane components of the network. Our vision for SOL stands in stark contrast to the state of affairs today, in which a developer faces programming a new SDN optimization either directly for a generic and low-level optimization solver such as `CPLEX` or using a heuristic algorithm designed by hand, after which she must translate the decision variables of the optimization to device configurations.

Path abstraction: For SOL to be useful and robust, we need a unifying abstraction that can capture the requirements of diverse classes of SDN network optimization applications described in the previous section. SOL is built using *paths* through a network as a core abstraction for expressing network optimization problems. This is contrary to how many optimizations are formulated in the literature — using a more standard edge-centric approach [1]. In our experience, however, an edge-centric approach forces complexity when presented with additional requirements, especially ones that attempt to capture path properties [29, 19].

In contrast, path-based formulations capture these requirements more naturally. For instance, much of the complexity in modeling service chaining or network function offloading applications from §2 is in capturing the path properties that need to be satisfied. With a path-based abstraction, we can simply define predicates that specify valid paths — e.g., those that include certain waypoints or that avoid a certain node (to anticipate that node's failure). In addition, we can model path-based resource use with ease. For example, usage of TCAM space in a switch corresponds to a traffic-carrying path traversing that switch (and thus a rule to accommodate that path). Without the path abstraction, modeling such constraints is difficult (cf., [39]). Finally, expressing constraints on nodes and edges does not introduce increased difficulty compared to edge-centric approach.

Scalability: In a pure flow-routing scenario, an edge-based formulation admits simple algorithms that guarantee polynomial-time execution. Path-based formulations, on the other hand, are often dismissed because of their inefficient appearance — after all, in the worst case, the number of paths in the network is exponential in the network size — or due to the complexity of algorithms to solve path based formulations (column-generation, decompositions, etc. [1]). However, in many practical scenarios, the number of valid paths (as defined by the application) is likely to be significantly smaller. Furthermore, multipath routing can provide only so much network diversity before its value diminishes [30]. So, the set of paths that need to be considered is not large.

SOL leverages an off-line path generation step to determine valid paths (step 1 of Fig. 6). Since for most applications, the set of valid paths is fairly static and does not need to be recomputed every time the optimization is run, we expect this step is infrequent. Next, SOL *selects* a subset of these paths (step 2) using a selection strategy (see §5) and runs the optimization with only the selected paths as input (step 3), to ensure that the optimization completes quickly. We show in §8 that this strategy still permits inclusion of sufficiently many paths for the optimization to converge to a (near) optimal value. So, while the efficiency of path-based optimization is a valid theoretical concern, in practice we show that there are practical heuristics to address this issue.

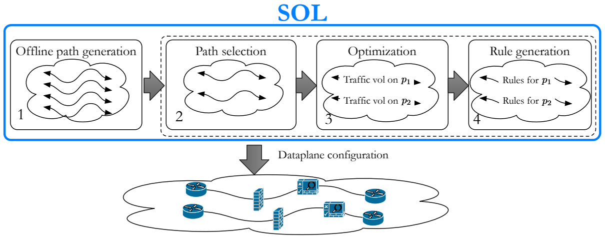


Figure 6: *SOL architecture, overview of the workflow*

Generating device configurations: SOL translates the decision variables from the SOL optimization to network device configurations to implement appropriate flow routing (step 4 of Fig. 6). The algorithm utilized in SOL to perform this translation is based on that in previous work [47, 20]. However, because the optimization is path-based, the algorithm is more straightforward and requires fewer steps.

`nodes`: Set of all nodes, part of the topology

`links`: Set of all links, part of the topology

`classes`: Set of all traffic classes

`paths`(c): Paths available for class $c \in$ `classes`; output by path-selection stage (§5)

Figure 7: ***Network data input***

	Var.	Description
Decision	$x_{c,p}$	Fraction of class-c flows allocated to path $p \in \texttt{paths}(c)$; non-integer
	b_p	Is path p used; binary
	b_v	Is node v used; binary
	b_l	Is link l used; binary
	$capvar^r_v$	Capacity allocated for resource r at node v; non-integer
Derived	a_c	Fraction of c's "demand" routed; non-integer
	$load^r_l$	Amount of resource r consumed by flows routed over link l; non-integer
	$load^r_v$	Amount of resource r consumed by flows routed via node v; non-integer

Figure 8: Variables internal to the optimization

4 SOL Detailed Design

In this section, we present the detailed design of SOL. We focus on the high-level API that the SDN application developer would use to express applications via SOL, and the impact of these API calls on the SOL's internal representation of the optimization problem. Note, however, that the developer "thinks" in terms of the high-level API rather than low-level details of dealing with the solver-level variables, how paths are identified, etc.

A developer begins a new optimization in SOL by instantiating an `opt` object via the `getOptimization` function and then building the optimization using *constraint templates*, which we explain below.

4.1 Preliminaries

Data inputs: There are two basic data inputs that the developer needs to provide to any network optimization. First, the network topology is a required input, specified as a graph with `nodes` and `links`. It also contains metadata of node/edge types or properties; e.g., nodes can have designated functions like "switch" or "middlebox". Second, SOL needs a specification of *traffic classes*, where each class c has associated ingress and egress nodes and some expected traffic volume. Each class can (optionally) be associated with a specification of the "processing" required for traffic in this class, e.g., service chaining. Finally, to each traffic class c is associated a set `paths`(c) available to route flows in class c; `paths`(c) is output by a path-selection preprocessing step described in §5.

Internal variables: SOL internally defines a set of variables summarized in Fig. 8. We reiterate that the developer does not need to reason about these variables and uses a high-level mental model as discussed earlier. There are two main kinds of variables:

- *Decision variables* that identify key optimization control decisions. The most fundamental decision variable is $x_{c,p}$, which captures traffic routing decisions and denotes the fraction of flow for a traffic class c that path $p \in \texttt{paths}(c)$ carries. This variable is central to various types of resource management applications as we will see later. To capture topological requirements (e.g., §2.3), we introduce three binary decision variables b_p, b_v, and b_l that denote whether each path, node or link (respectively) is enabled ($= 1$) or disabled ($= 0$). The variable $capvar^r_v$ is the SOL-assigned allocation of resource-r to node v.
- *Derived variables* are functions defined over the above decision variables that serve as convenient "shorthands". a_c denotes the total fraction of flow for class c that is carried by all paths. The load variables $load^r_v$ and $load^r_l$ model the consumption of resource r on node v and link l, respectively.

There are low-level API calls [1] that return the names of these internal variables, which can be used to access each one's value in a public map of names to values, if needed.

4.2 Routing requirements

Routing constraints control the allocation of flow in the network. `addAllocateFlowConstraint` creates the necessary structure for routing the traffic through a set of paths for each traffic class. Some network applications try to satisfy as much of their flow demands as possible (e.g., max-flow) while others (e.g., TE) want to "saturate" demands. For example, a developer of a TE application (§2.1) would like to route all traffic though the network, and thus she would add the following high-level routing constraint templates to her empty `opt` object:

```
opt.addAllocateFlowConstraint()
opt.addRouteAllConstraint()
```

In contrast, a simple max-flow would only need `addAllocateFlowConstraint` since there is no requirement on saturating demands in that case.

The `addEnforceSinglePath`(C) constraint forces a single flow-carrying path per class $c \in C$, preventing flow-splitting and multipath routing.

Internals: `addAllocateFlowConstraint` ensures that the total traffic flow across all chosen paths for the class c matches the variable a_c.

$$\forall c \in \texttt{classes} : \sum_{p \in \texttt{paths}(c)} x_{c,p} = a_c$$

[1] We also expose low-level APIs (see Appendix A) for advanced users.

Group	Function	Description
Routing ($C \subseteq$ `classes`)	`addAllocateFlowConstraint`	Allocate flow in the network
	`addRouteAllConstraint`	Route all traffic demands
	`addEnforceSinglePath` (C)	For each $c \in C$, at most one $p \in \texttt{paths}(c)$ is enabled.
Capacities	`addLinkCapacityConstraint` (r, *lnCap*, `linkCapFn`)	If l is in *lnCap*, then limit utilization of link resource r on link l to $lnCap[l]$.
	`addNodeCapacityConstraint` (r, *ndCap*, `nodeCapFn`)	If v is in *ndCap*, then limit utilization of node resource r on node v to $ndCap[v]$.
	`addNodeCapacityPerPathConstraint` (r, *ndCap*, `nodeCapFn`)	If v is in *ndCap*, then limit utilization of node resource r on node v by enabled paths to $ndCap[v]$.
	`addCapacityBudgetConstraint` (r, N, *totCap*)	Limit total type-r resources allocated to nodes in $N \subseteq$ `nodes` to *totCap*. Used when SOL is allocating capacities.
Topology control ($C \subseteq$ *classes*)	`addRequireAllNodesConstraint` (C)	For each $c \in C$ and each $p \in \texttt{paths}(c)$, p can be enabled iff all nodes on p are enabled.
	`addRequireSomeNodesConstraint` (C)	For each $c \in C$ and each $p \in \texttt{paths}(c)$, p can be enabled iff some node on p is enabled.
	`addRequireAllEdgesConstraint` (C)	For each $c \in C$ and each $p \in \texttt{paths}(c)$, p can be enabled iff all links on p are enabled.
	`addPathDisableConstraint` (C)	For each $c \in C$ and each $p \in \texttt{paths}(c)$, p can carry traffic only if it is enabled.
	`addBudgetConstraint` (`nodeBudgetFn`, k)	Total cost of enabled nodes, as computed using `nodeBudgetFn`, is at most k.
Objective	`setPredefinedObjective` (*name*)	Set one of the predefined functions as the objective (see Fig. 11).

Figure 9: ***Selected constraint template functions for building optimizations; see Fig. 10 for `linkCapFn`, `nodeCapFn`, and `nodeBudgetFn`***

Similarly, `addRouteAllConstraint` implies:

$$\forall c \in \texttt{classes}: \; a_c = 1$$

Due to space limitations, we do not provide the formal basis for `addEnforceSinglePath`.

4.3 Resource capacity constraints

As we saw in §2, SDN optimizations have to deal with a variety of capacity constraints for network resources such as link bandwidth, switch rules, and middlebox CPU and memory. SOL allows users to write custom resource management logic by specifying several "cost" functions, depicted in Fig. 10. These functions prescribe how to compute the cost of routing traffic through a link, a node, or a given path. SOL provides default implementations of these for common tasks, but allows the user to specify their own logic, as well, as we will show later (§6).

These cost functions can then be passed into constraint templates. For example, to add a constraint that limits link usage, the user can invoke the template function `addLinkCapacityConstraint` with a resource that we are constraining (e.g., 'bandwidth'), a map of links to their capacities,[2] and optionally, a custom `linkCapFn` to compute the cost of traffic on a link.

```
opt.addLinkCapacityConstraint ('bandwidth',
    {(1,2): 10**7, (2,3): 10**7},
    defaultLinkFunction)
```

This indicates that bandwidth should not exceed 10 Mbps for links 1-2 and 2-3. Note that the default function is purely for illustration; the developer can write her own `linkCapFn` (recall Fig. 10).

`addNodeCapacityPerPathConstraint` generates constraints on the nodes that do not depend on the traffic, but rather on the routing path. That is, the cost of routing at a node does not depend on the volume or type of traffic being routed; it depends on the path and its properties. The best example of such usage is accounting for the limited rule space on a network switch (e.g., §2.2). If a path is "active", the rule must be installed on each switch to support the path.

Internals: `addLinkCapacityConstraint` and `addNodeCapacityConstraint` rely on `linkCapFn` and `nodeCapFn`, respectively, to compute the cost of using a particular resource at a link or node if all of the class-c traffic was routed to it. Internally, the load is multiplied by the $x_{c,p}$ variable to capture the load accu-

[2] When capacities should be allocated by the optimization itself, a capacity of TBA (meaning To Be Allocated) can be specified, instead.

`linkCapFn`(l,c,p,r): Amount of resource type r consumed if all class-c traffic is allocated to path $p \ni l$ for link l

`nodeCapFn`(v,c,p,r): Amount of resource r consumed if all class-c traffic is allocated to path $p \ni v$ for node v

`nodeBudgetFn`(v): Cost of using node v; required with `addBudgetConstraint`

`routingCostFn`(p): Cost of routing along path p; required with `minRoutingCost`

`predicate`(p): Determine whether any given path is valid by returning `True` or `False`

Figure 10: ***Customizable functions***

rately, then the load is capped by a user-provided *lnCap* (*ndCap*), which is a mapping of links (nodes) to capacities for a given resource r. (Similar node capacity equations not shown for brevity.)

$$\begin{aligned}&\forall l \text{ in } lnCap:\\ &\quad load_l^r = \sum_c \sum_{p \in \texttt{paths}(c): l \in p} x_{c,p} \times \texttt{linkCapFn}(l,c,p,r)\\ &\quad load_l^r \leq lnCap[l]\end{aligned}$$

The `addNodeCapacityPerPathConstraint` functions a bit differently, as it depends on enabled paths:

$$\begin{aligned}&\forall v \text{ in } ndCap:\\ &\quad load_v^r = \sum_c \sum_{p \in \texttt{paths}(c): v \in p} b_p \times \texttt{nodeCapFn}(v,c,p,r)\\ &\quad load_v^r \leq capvar_v^r\\ &\quad \text{if } ndCap[v] \neq \texttt{TBA} \text{ then } capvar_v^r = ndCap[v]\end{aligned}$$

4.4 Node/link activation constraints

Next set of constraints, when used, allow developers to logically model the act of *enabling* or *disabling* nodes, links, and paths; e.g., for managing energy or other costs (e.g., §2.3). We identify two possible modes of interactions between these topology modifiers, and the optimization developer can choose the one that is most suitable for their context. 1) `addRequireAllNodesConstraint` captures the property that disabling a node disables all paths that traverse it; and 2) `addRequireSomeNodesConstraint` captures the property that enabling a node permits any path traversing it to be enabled, as well. The latter version is suitable when, e.g., a node can still route traffic even if its other (middlebox) functionality is disabled, and so a path containing that node is potentially useful as providing middlebox functions if at least one of its nodes is enabled. There are analogous constraint templates for links, but we omit them here for brevity. A third constraint template, `addPathDisableConstraint`, restricts a path to carry traffic only if it is enabled.

For example, a developer trying to implement the application from §2.3 can model the requirements for shutting off datacenter nodes by adding the `addRequireAllNodesConstraint` and `addPathDisableConstraint` templates:

```
opt.addRequireAllNodesConstraint (trafficClasses)
opt.addPathDisableConstraint (trafficClasses)
```

Other efficiency considerations may enforce a budget on the number of enabled nodes, to model constraints on total power consumption of switches/middleboxes, cost and budget of installing/upgrading particular switches, etc. These are captured via the `addBudgetConstraint` template function.

Internals: Internally, these topology modification templates are achieved using the binary variables we introduced earlier. Specifically, the above requirements can be formalized as follows:

	$\forall p \in \texttt{paths}(c):$
`addRequireAllNodesConstraint`	$\forall v \in p:\ b_p \leq b_v$
`addRequireSomeNodesConstraint`	$b_p \leq \sum_{v \in p} b_v$
`addPathDisableConstraint`	$x_{c,p} \leq b_p$

Naturally, similar constraints are constructed for links. Note that `addPathDisableConstraint` is crucial to the correctness of the optimization in that it enforces that no traffic traverses a disabled path. For brevity, we do not provide the formal equations for `addBudgetConstraint`.

4.5 Specifying network objectives

The goal of SDN applications is eventually to optimize some network-wide objective, e.g., maximizing the network throughput, balancing load, or minimizing total traffic footprint. Fig. 11 lists the most common objective functions, drawing on the applications considered in §2. For instance, the developer of a TE application may want to implement the objective of minimizing the maximum link load and thus add the following code snippet:

```
opt.setPredefinedObjective (minMaxLinkLoad,
    'bandwidth')
```

Other optimizations (e.g., §2.4) may need to minimize the total routing cost and include a `minRoutingCost` objective. This objective is parameterized with `routingCostFn`(p); i.e., developers can plugin their own cost metrics such as number of hops or link weights. As shown, we also provide a range of natural load-balancing templates. SOL also exposes a low-level API for specifying other complex objective functions, which we describe in Appendix A.

5 Path generation and selection

Given these constraint templates, the remaining question is how we populate the path set `paths`(c) for each

`maxAllFlow`	maximize $\sum_{c \in \texttt{classes}} a_c$
`minMaxNodeLoad` (r)	minimize $\max_{v \in \texttt{nodes}} load_v^r$
`minMaxLinkLoad` (r)	minimize $\max_{l \in \texttt{links}} load_l^r$
`minRoutingCost`	$\sum_{c,p} \texttt{routingCostFn}(p) \times x_{c,p}$

Figure 11: ***Common objective functions***

traffic class c to meet two requirements. First, each $p \in \texttt{paths}(c)$ should satisfy the desired policy specification for the class c. Second, $\texttt{paths}(c)$ should contain paths for each class c that make the formulation tractable and yet yield near-optimal results. We describe how we address each concern next.

Generation: First, to populate the paths, SOL does an offline enumeration of all simple (i.e., no loops) paths per class.[3] Given this set, we filter out the paths that do not satisfy the user-defined predicate `predicate`, i.e., where $\texttt{predicate}(p) = \texttt{True}$ only if p is a valid path. (We can generalize this to allow different predicates per class; not shown for brevity.)

In practice, we implement the predicate as a flexible `Python` callable function rather than constrain ourselves to specific notions of path validity (e.g., regular expressions as in prior work [45]). Using this predicate gives the user flexibility to capture a range of possible requirements. Examples include waypoint enforcement (forcing traffic through a series of middleboxes in order); enforcing redundant processing (e.g., through multiple IDS, in case one fails open); and limiting network latency by mandating shorter paths.

Selection: Using all valid paths per class may be inefficient since the number of paths grows exponentially with the size of the network, meaning that the LP/ILP that SOL generates will quickly become too large to solve in reasonable time. SOL thus provides path selection algorithms that choose a subset of valid paths (number of paths denoted as `selectNumber`) that are still likely to yield near-optimal results in practice. Specifically, two natural methods work well across the spectrum of applications we have considered: (1) shortest paths for latency-sensitive applications (`selectStrategy = shortest`) or (2) random paths for applications involving load balancing (`selectStrategy = random`). SOL is flexible to incorporate other selection strategies, e.g., picking paths with minimal node overlap for fault tolerance. We find `random` works well for many applications that require load balancing. We conjecture this is because choosing random paths on sufficiently rich topologies yields a high degree of edge-disjointedness among the chosen paths, yielding sufficient degrees of freedom for balancing loads.

```
SIMPLE_predicate = functools.partial(waypointMboxPredicate
      , order=('fw','ids'))
def SIMPLE_NodeCapFunc(node,tc,path,resource,nodeCaps):
  if resource=='cpu' and node in nodeCaps['cpu']:
    return tc.volFlows*tc.cpuCost/nodeCaps[resource][node]

capFunc = functools.partial(SIMPLE_NodeCapFunc, nodeCaps=
      nodeCaps)

def SIMPLE_TCAMFunc(node, tc, path, resource):
  return 1
# Path generation, typically run once in a precomputation
      phase
opt = getOptimization()
pptc = generatePathsPerTrafficClass(topo, trafficClasses,
      SIMPLE_predicate, 10, 1000,
       functools.partial(useMboxModifier, chainLength=2))
# Allocate traffic to paths
pptc = chooserand(pptc, 5)
opt.addDecisionVariables(pptc)
opt.addBinaryVariables(pptc, topo, ['path','node'])
opt.addAllocateFlowConstraint(pptc)
opt.addRouteAllConstraint(pptc)
opt.addLinkCapacityConstraint(pptc, 'bandwidth', linkCaps,
       defaultLinkFuncNoNormalize)
opt.addNodeCapacityConstraint(pptc, 'cpu',
      {node: 1 for node in topo.nodes() if 'fw' or
      'ids' in topo.getServiceTypes(node)}, capFunc)
opt.addNodeCapacityPerPathConstraint(pptc, 'tcam',
      nodeCaps['tcam'], SIMPLE_TCAMFunc)
opt.setPredefinedObjective('minmaxnodeload','cpu')
opt.solve()
obj = opt.getSolvedObjective()
pathFractions = opt.getPathFractions(pptc)
c = controller()
c.pushRoutes(c.getRoutes(pathFractions))
```

Figure 12: ***Code to express SIMPLE [39] in SOL***

Developer API: The developer can specify the path predicate and selection strategy, but she does not need to be involved in the low-level details of generation and selection. SOL also provides APIs for developers to add their own logic for generation and selection; we do not discuss these due to space limitations.

6 Examples

Next, we show end-to-end examples to highlight the ease of using the SOL APIs to write existing and novel SDN network optimizations. These examples are actual `Python` code that can be run, not just pseudocode. By comparison, the code is significantly higher-level and more readable than the equivalent `CPLEX` code would be, as it does not need to deal with large numbers of underlying variables and constraints.

Service chaining (§2.2): As a concrete instance of the service chaining example, we consider SIMPLE [39]. SIMPLE involves the following requirements: route all traffic through the network, enforce the service chain (e.g., "firewall followed by IDS") policy for all traffic, load balance across middleboxes, and do so while respecting CPU, TCAM, and bandwidth requirements. Fig. 12 shows how the SIMPLE optimization can be written in ≈ 25 lines of code. This listing assumes that topology and traffic classes have been set up, in the `topo` and

[3] This is to simplify the forwarding rules without resorting to tunneling or packet tagging [39].

trafficClasses objects, respectively.

The first part of the figure shows function definitions and the path generation step, which would typically be performed once as a precomputation step. We start by defining a path predicate (line 1) for basic enforcement through middleboxes by using the SOL-provided function with the middlebox order. The next few lines (lines 2–4) show a custom node capacity function to normalize the CPU load between 0 and 1. This computes the processing cost per traffic class (number of flows times CPU cost) normalized by the current node's capacity. Similarly, the TCAM capacity function captures that each path consumes a single rule per switch (line 7). The user gets the optimization object (line 10), and generates the paths (line 11), obtaining the "paths per traffic class" (pptc) object. The path generation algorithm is parameterized with the custom SIMPLE_predicate, a limit on path length of 10 nodes, and a limit on the number of paths per class of 1000. It is also instructed to evaluate every possible use of two middleboxes on a routing path for inclusion as a distinct path in the output.

The remaining lines show what would be executed whenever a new allocation of traffic to paths is desired. Line 13 selects 5 random paths per traffic class; lines 14–20 add the routing and capacity constraints. We use the default link capacity function for bandwidth constraints, and our own functions for CPU and TCAM capacity. Because the CPU capacity function normalizes the load, the capacity of each node is now 1 (line 19). The program selects a predefined objective to minimize the CPU load (line 21) and calls the solver (line 22). Finally, the program gets the results and interacts with the SDN controller to automatically install the rules (line 26).

ElasticTree [19]: Due to space limitations we only show the most important differences between ElasticTree and SIMPLE. There is no requirement on paths, and so nullPredicate is used for path generation. We use link binary variables (see line 1 below) and the node/link activation constraints (lines 2–4). Finally, we use the low-level API (see App. A) to define power consumption for switches and links (lines 5, 6, wherein "opt.bn(node)" and "opt.be(u,v)" retrieve the names of variables $b_{\texttt{node}}$ and $b_{(\texttt{u,v})}$ from Fig. 8, respectively) and use these variables to define a custom objective function (line 7).

```
1 opt.addBinaryVariables(pptc,topo,['path','node','edge'])
2 opt.addRequireAllNodesConstraint(pptc)
3 opt.addRequireAllEdgesConstraint(pptc)
4 opt.addPathDisableConstraint(pptc)
5 opt.defineVar('SwitchPower', {opt.bn(node):switchPower[
      node] for node in topo.nodes()})
6 opt.defineVar('LinkPower', {opt.be(u, v): linkPower[(u, v
      )] for u, v in topo.links()})
7 opt.setObjectiveCoeff({'SwitchPower': .75, 'LinkPower':
      .25}, 'min')
```

We refer the reader to Appendix B for other examples that include new and more complex applications.

7 Implementation

Developer interface: We currently provide a Python API for SDN optimization that is an extended version of the interface described in §4.

Invoking solvers: We use CPLEX (via its existing Python API) as our underlying solver. This choice largely reflects our familiarity with the tool, and we could substitute CPLEX with other solvers like Gurobi. SOL offers APIs to exploit solver capabilities to use a previously computed solution and incrementally find a new solution. This approach is typically faster than starting from scratch and so is useful for faster reconfigurations. SOL also allows hard-limiting of the optimization runtime, albeit affecting the optimality of the solution.

Path generation: Path generation is an inherently parallelizable process; we simply launch separate Python processes for different traffic classes. We currently support two path selection algorithms: random and shortest. It is easy to add more algorithms as new applications emerge.

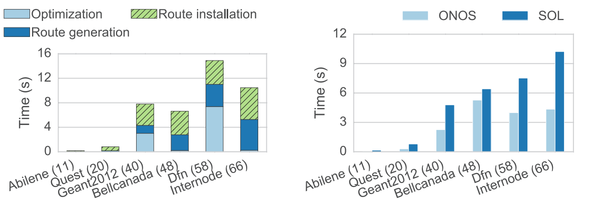
Rule generation and control interface: We implement applications for ONOS [5] and use custom REST API to allow remote batch installation of the relevant rules. We generate the rules based on the optimization output, using network prefix splitting to implement the fractional responsibilities represented by the $x_{c,p}$ variables. This step is similar to prior work that map fractional processing and forwarding responsibilities onto network flows (e.g., [47, 20]), and so we do not repeat it here. With ONOS, we leverage *path intents* [5]: while not required, it facilitates easier integration.

Minimizing reconfiguration changes: Networks are in flux during reconfigurations with potential performance or consistency implications, and thus it is desirable to minimize unnecessary configuration changes. SOL supports constraints that bound (or minimize) the logical distance between a previous solution and the new solution to help minimize the number of flows that have to be assigned a new route. In this way, SOL supports path selection that gives priority to previously selected paths.

8 Evaluation

In this section we show that SOL

- performs well with the ONOS controller;
- computes optimal solutions for published applications order(s) of magnitude faster than their original optimizations; allows to minimize traffic churn
- is either faster or has richer functionality than state-of-the-art related work;



(a) ***Time for SOL to configure the network using the*** `ONOS` ***controller for a traffic engineering application.***

(b) ***Route generation & installation time of SOL traffic engineering app vs.*** `ONOS` ***all-pair shortest paths***

Figure 13: ***Deployment benchmarks using the*** `ONOS` ***controller***

- significantly reduces development effort in comparison to manually coding optimization applications; and
- scales well, because it computes near-optimal solutions using few paths per traffic class.

Setup: We evaluate the effect of using SOL to implement three existing SDN applications: ElasticTree [19], SIMPLE [39], and Panopticon [29]. For each application, we implemented the original formulation presented in prior work or obtained the original source code. We refer to these as "original" formulations (and solutions). We chose topologies of various sizes from the TopologyZoo dataset [28]; when indicating a topology, we generally include the number of nodes in the topology in parentheses, e.g., "Abilene (11)" for the 11-node Abilene topology. For ElasticTree, we also constructed FatTree topologies of various sizes [2]. We synthetically generated traffic matrices using a uniform traffic matrix for the FatTree networks and a gravity-based model [43] for the TopologyZoo topologies. We used randomly sampled values from a log-normal distribution as "populations" for the gravity model. Unless otherwise specified, we used 5 paths per traffic class when running SOL. All times below refer to computation on computers with 2.4GHz cores and 128GB of RAM. For deployment benchmarks, we used the default Mininet [31] virtual machine to emulate topologies.

8.1 Deployment benchmarks

We setup a variety of emulated networks using `Mininet` and `ONOS`. We measured time for SOL to run the optimization for a traffic engineering goal and compute and install network routes. Fig. 13a shows the times to perform each step. SOL exhibits low optimization and route generation times, making route installation the most time-consuming part of the configuration process. This bottleneck is caused by the number of rules that must be installed and by the controller platform. Fig. 13b shows the time to generate and install routes for a traffic engineering application using SOL, in contrast to installing shortest path routes using methods available in `ONOS`. The difference is insignificant, and exists due to the additional optimization time and because of the multiple paths per source-destination pair in the SOL case.

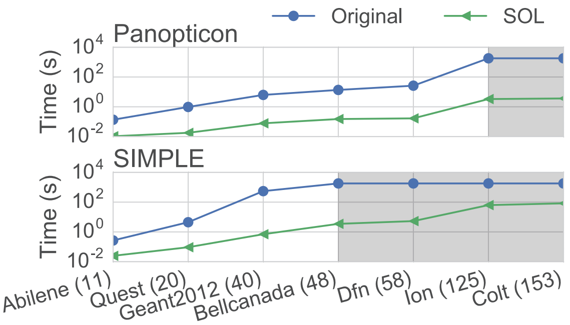


Figure 14: ***Optimization runtime of SOL and original formulations; gray regions show where original formulation could not be solved within 30 mins***

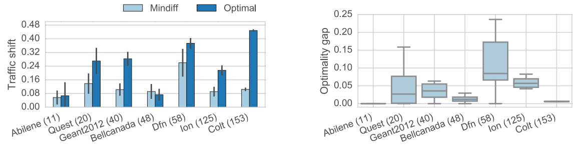
8.2 Optimality and scalability

Comparing to optimal: Next, we examine how well SOL's results match original solutions, which are optimal (by definition). In all cases except ElasticTree, SOL finds the optimal solution. Due to complexity of ElasticTree's optimization, SOL suffers a 10% optimality gap: the relative error in the objective value computed by SOL (i.e., relative to the true optimal objective value).

SOL solution times are at least one order of magnitude faster than solving the original formulations, and are often two or even three orders of magnitude faster. Fig. 14 shows run times to find original solutions. The runtime was capped at 30 min (1800 s), after which the execution was aborted. Several original formulations did not complete in that time, such as SIMPLE for topologies Bellcanada and larger, and Panopticon for Ion and larger. The topologies for which original solutions could not be found are indicated in the gray regions in Fig. 14.

Comparing to specialized heuristics: We found that SOL performs fairly well even compared to specialized heuristics. Specifically, we compared the performance of SOL to the custom heuristic for SIMPLE, obtained from its authors. The runtime of SOL is comparable to that of the SIMPLE heuristic algorithm, with a performance gap of at most 3 seconds on the largest topologies we considered (up 58 nodes, namely the "Dfn" topology). We believe the benefit of not having to design custom heuristics outweighs this performance gap.

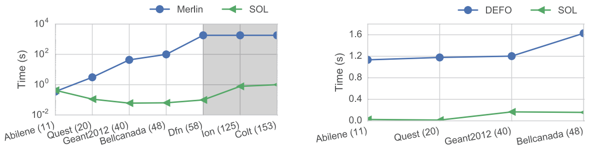
Responding to traffic changes: We explore the benefits of the reconfiguration minimization capabilities of SOL, for simplicity dubbed "mindiff." We first computed an optimal solution for a traffic engineering application; then, a random permutation of the traffic matrix triggered the re-computation with mindiff enabled. When computing the new solution, we assigned 4× greater priority to the TE objective than the mindiff objective. Fig. 15a shows that with mindiff enabled, up to an additional 35%



(a) ***Fraction of traffic reassigned to different paths with and without "mindiff"***

(b) ***Optimality gap when using "mindiff"***

Figure 15: ***Traffic shift and optimality gap when using reconfiguration minimization capabilities of SOL***



(a) ***Optimization runtimes of SOL and Merlin; gray region indicates where Merlin did not complete in 30 mins***

(b) ***Optimization runtimes of SOL and DEFO, for a traffic engineering application***

Figure 16: ***Runtime of SOL vs. state-of-the-art optimization frameworks***

of *total* traffic stays on its original paths across reconfigurations, versus being reassigned to new paths by the optimal solution. Naturally, SOL sacrifices some optimality in the original TE objective (shown in Fig. 15b).

8.3 Comparison to Merlin and DEFO

Merlin [45] tackles problems of network resource management similar to SOL. While the goals and formulations of Merlin and SOL are quite different, we use this comparison to highlight the generality of SOL and the power of its path abstraction. Specifically, Merlin uses a more heavyweight optimization that is always an ILP and operates on a graph that is substantially larger than the physical network. We implemented the example application taken from the Merlin paper using both SOL and Merlin. Fig. 16a shows that SOL outperforms Merlin by two or more orders of magnitude.

DEFO [18] is an optimization framework that aims to simplify traffic engineering [18]. We obtained the DEFO authors' implementation and compared the optimization times of DEFO and SOL on a simple traffic engineering application. DEFO and SOL exhibit comparable runtimes (see Fig. 16b). However, DEFO lacks the ability to express more complex applications and objectives and to filter paths by arbitrary predicates.

8.4 Developer benefits

SOL is a much simpler framework for encoding SDN optimization tasks, versus developing custom solutions by hand. In an effort to demonstrate this simplicity somewhat quantitatively, Table 1 shows the number of lines of code (LOC) in our SOL implementations of various applications ("SOL lines of code"), and the ratio of the LOC of the original formulations to the LOC for our SOL implementations ("Estimated improvement"). We acknowledge that lines-of-code comparisons are inexact, but we do not know of other ways of comparing "development effort" without conducting user studies.

Name	SOL lines of code	Estimated improvement
ElasticTree	16	21.8×
Panopticon	13	25.7×
SIMPLE	21	18.6×

Table 1: ***Development effort benefits provided by SOL***

We believe that the improvements in Table 1 are conservative. First, producing original formulations is a much more complex and delicate process than writing SOL code. We primarily attribute this difference to needing to account for `CPLEX` (or other solvers, e.g., [17, 33]) particulars at all; with SOL, these particulars are completely hidden from the developer. Second, SOL translates its optimization results to device configurations, whereas this functionality is not even included in our scripts for producing original formulations. Producing device configurations from original solutions would require designing an extra algorithm to map the variables in each formulation to relevant device configurations.

8.5 Sensitivity

SOL solutions require the specification of both the number (`selectNumber`) and type (`shortest` or `random`) of paths to select per traffic class. In this section, we quantify how sensitive SOL is to these parameters.

Number of Paths: Fig. 17 shows the SOL's runtime and optimality gap as a function of the number of paths per class for two applications: SIMPLE and Panopticon. Unsurprisingly, with a larger number of paths, SOL's runtime increases. However, this is not a significant concern, since we find optimal solutions at `selectNumber` as low as 5. These numbers are representative of all applications and topologies we have considered.

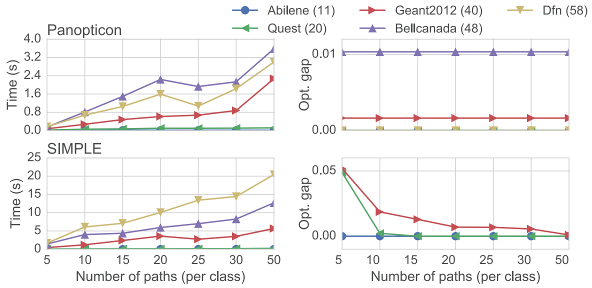


Figure 17: ***Runtime and optimality gap as function of paths; optimality is achieved in most cases with as few as 5 paths per class***

Path selection strategy: We evaluated different selection strategies across topologies and applications (omitted for brevity). Our results were consistent with our experiences more generally that most problems lend themselves to a fairly obvious path selection strategy: those with need for load balancing are likely to benefit from random and those that are latency-sensitive benefit from shortest. If in doubt, however, both strategies can be attempted.

Path selection and generation costs: Since path selection is part of the optimization cycle, we ensure that path selection times are small, ranging from 0.1 to 3 seconds across topologies. Path selection is preceded by a path generation phase that enumerates the simple paths per class. Path generation is moderately costly for large topologies, e.g., taking <300 s for the largest presented topology, when parallelized to 60 threads. However, we emphasize that path generation can be relegated to an offline precomputation phase that is only performed once.

9 Discussion

Expressiveness of SOL: We make no claim that SOL is a panacea, capable of expressing *any* optimization, nor do we have a formal way to decide if a problem fits the SOL paradigm. We can provide guidelines as to which problems are well-suited (or not) for SOL. Optimizations with complex path predicates benefit greatly from SOL, as path generation and validation is performed offline, not during optimization. So do problems with resource constraints dependent on paths (e.g., SIMPLE with its TCAM constraints). Problems with no path predicates and very large interconnected topologies (i.e., large datacenter networks) are less likely to benefit from SOL. However, we plan to explore alternative approaches (e.g., hierarchical optimization) approaches to provide benefits in that space as well.

Analytical guarantees: While our empirical results suggest that random or shortest paths yield near-optimal solutions with a small selectNumber, they also raise interesting theoretical questions: can we prove that these selection strategies permit near-optimal solutions for specific classes of problems?

Very large/dynamic networks: For very large networks (>100 nodes) SOL might not perform as well as heuristic solutions, especially for applications that require an ILP, since the number of paths grows very large. In such cases, we can utilize general approximation heuristics, such as randomized rounding, to maintain its ability to react to network changes quickly.

Composition: Having a unified optimization layer atop SDN controllers exposes opportunities to compose applications. We plan to explore these in future work.

10 Related Work

We already discussed the optimization applications that motivated SOL. Here we focus on other related work.

Higher-layer abstractions for SDN: This work includes new programming languages (e.g., [41, 12]), testing and verification tools (e.g, [27]), semantics for network updates (e.g., [42]), compilers for rule generation (e.g., [26]), abstractions for handling control conflicts (e.g., [4]), and APIs for users to express requirements (e.g., [10]). These works do not address the optimization component, which is the focus of SOL.

Languages for optimization: There are several modeling frameworks such as AMPL [13], Mosek [33], PyOpt [37], and PuLP [32] for expressing optimization tasks. However, these do not specifically simplify network optimization. SOL is a domain-specific library that operates at a higher level of semantics than these "wrappers". SOL offers a path-based abstraction for writing network optimizations, exploits this structure to solve these optimizations quickly, and generates network device configurations that implement its solutions.

Network resource management: Merlin is a language for network resource management [45]. In terms of the applications that it can support, Merlin is restricted to using path predicates expressed as regular expressions. Our experiments suggest that SOL is three orders of magnitude faster than Merlin using the same underlying solvers. That said, Merlin's "language-based" approach provides other capabilities (e.g., verified delegation) that SOL does not (try to) offer. DEFO is another optimization framework that focuses on traffic engineering and service chaining applications [18]. Their goal is not to develop a general framework, but rather to support easy management of carrier-grade networks, which they accomplish using a two-layer architecture and support for networks that are not OpenFlow-enabled via segment routing. Other works focus on traffic-steering optimization (e.g., [39, 7]). SOL offers a unifying abstraction that covers many network management applications.

11 Conclusion

While network optimization is central to many SDN applications, few efforts attempt to make it accessible. Our vision is a general, efficient framework for expressing and solving network optimizations. Our framework, SOL, achieves both generality and efficiency via a path-centric abstraction. We showed that SOL can concisely express applications with diverse goals (traffic engineering, offloading, topology modification, service chaining, etc.) and yields optimal or near-optimal solutions with often better performance than custom formulations. Thus, SOL can lower the barrier to entry for novel SDN network optimization applications.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Ankit Singla, for helpful comments and guidance. This work was supported in part by grant N00014-13-1-0048 from the Office of Naval Research, NSF grants 1535917 and 1536002, an NSF Graduate Research Fellowship, the Science of Security Lablet at North Carolina State University, and by Intel Labs' University Research Office.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice hall, 1993.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74, 2008.

[3] M. Allalouf and Y. Shavitt. Centralized and distributed algorithms for routing and weighted max-min fair bandwidth allocation. *ACM/IEEE Transactions on Networking*, 16(5):1015–1024, 2008.

[4] A. AuYoung, S. Banerjee, J. Lee, J. C. Mogul, J. Mudigonda, L. Popa, P. Sharma, and Y. Turner. Corybantic: Towards the modular composition of SDN control programs. In *ACM HotNets*, 2013.

[5] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. ONOS: towards an open, distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[6] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral. Deep packet inspection as a service. In *ACM CoNEXT*, pages 271–282, 2014.

[7] Z. Cao, M. Kodialam, and T. Lakshman. Traffic steering in software defined networks: planning and online routing. In *ACM SIGCOMM Workshop on Distributed Cloud Computing*, pages 65–70, 2014.

[8] M. Charikar, Y. Naamad, J. Rexford, and K. Zou. Multi-Commodity Flow with In-Network Processing. Manuscript, www.cs.princeton.edu/~jrex/papers/mopt14.pdf.

[9] E. Danna, S. Mandal, and A. Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *IEEE Conference on Computer Communications*, pages 846–854, 2012.

[10] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An API for application control of SDNs. In *ACM SIGCOMM*, August 2013.

[11] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *IEEE Conference on Computer Communications*, volume 2, 2000.

[12] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291, 2011.

[13] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A mathematical programming language*. AT&T Bell Laboratories Murray Hill, 1987.

[14] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella. Stratos: A network-aware orchestration layer for virtual middleboxes in clouds. *arXiv preprint arXiv:1305.0209*, 2013.

[15] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *ACM SIGCOMM*, 2014.

[16] G. Gibb, H. Zeng, and N. McKeown. Outsourcing network functionality. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2012.

[17] Gurobi. http://www.gurobi.com/.

[18] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *ACM SIGCOMM*, 2015.

[19] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 19–21, 2010.

[20] V. Heorhiadi, S. K. Fayaz, M. K. Reiter, and V. Sekar. SNIPS: A software-defined approach for scaling intrusion prevention systems via offloading. In *10th International Conference on Information Systems Security*, Dec. 2014.

[21] V. Heorhiadi, M. K. Reiter, and V. Sekar. New opportunities for load balancing in network-wide intrusion detection systems. In *ACM CoNEXT*, 2012.

[22] V. Heorhiadi, M. K. Reiter, and V. Sekar. SOL bitbucket repository. https://bitbucket.org/progwriter/sol/, 2015.

[23] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, pages 15–26, 2013.

[24] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, pages 3–14, 2013.

[25] X. Jin, L. E. Li, L. Vanbever, and J. Rexford. Softcell: Scalable and flexible cellular core network architecture. In *ACM CoNEXT*, 2013.

[26] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the one big switch abstraction in software-defined networks. In *ACM CoNEXT*, pages 13–24, 2013.

[27] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[28] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765 –1775, October 2011.

[29] D. Levin, M. Canini, S. Schmid, F. Schaffert, A. Feldmann, et al. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In *USENIX Annual Technical Conference*, 2014.

[30] X. Liu, S. Mohanraj, M. Pioro, and D. Medhi. Multipath routing from a traffic engineering perspective: How beneficial is it? In *IEEE International Conference on Network Protocols*, pages 143–154, 2014.

[31] Mininet. http://mininet.org/.

[32] S. Mitchell, M. O'Sullivan, and I. Dunning. Pulp: a linear programming toolkit for python, 2011.

[33] Mosek. https://mosek.com/.

[34] Network functions virtualisation – introductory white paper. http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[35] Opendaylight SDN controller. http://www.opendaylight.org/.

[36] S. Palkar, C. Lan, S. Han, K. Jang, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A runtime framework for network functions. In *ACM Symposium on Operating Systems Principles*, 2015.

[37] R. E. Perez, P. W. Jansen, and J. R. R. A. Martins. pyOpt: A Python-based object-oriented framework for nonlinear constrained optimization. *Structures and Multidisciplinary Optimization*, 45(1):101–118, 2012.

[38] M. Pióro, P. Nilsson, E. Kubilinskas, and G. Fodor. On efficient max-min fair routing algorithms. In *Computers and Communication*, pages 365–372. IEEE, 2003.

[39] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *ACM SIGCOMM*, 2013.

[40] S. Raza, G. Huang, C.-N. Chuah, S. Seetharaman, and J. P. Singh. Measurouting: a framework for routing assisted traffic monitoring. *ACM/IEEE Transactions on Networking*, 20(1):45–56, 2012.

[41] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular SDN programming with pyretic. *;login: Magazine*, 38(5):128–134, 2013.

[42] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM*, 2012.

[43] M. Roughan. Simplifying the synthesis of internet traffic matrices. *ACM SIGCOMM Computer Communication Review*, 35, 2005.

[44] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *ACM SIGCOMM*, 2012.

[45] R. Soule, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *ACM CoNEXT*, 2014.

[46] N. Vasić, P. Bhurat, D. Novaković, M. Canini, S. Shekhar, and D. Kostić. Identifying and using energy-critical paths. In *ACM CoNEXT*, 2011.

[47] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Hot-ICE*, 2011.

A Advanced users and low-level interface

While the SOL API described in §4 is general and expressive enough to capture the diverse requirements of the broad spectrum of applications, we also expose a low-level API that gives more control to the user by giving access to the SOL internal variables. Advanced users can use this API for further customization.

For instance, API calls enable the names of the internal variables in Fig. 8 to be retrieved and their values determined. Similarly, using the `defineVar` (*name*, *coeffs*, *lb*, *ub*) function, the user can create a new variable with name *name*, specify numeric lower and upper bounds (*lb* and *ub*), and equate it to a linear combination of any other existing variables as specified by *coeffs*, a map from variable names to numeric coefficients. This is a useful primitive when specifying complex objectives. SOL also allows setting a custom objective function that is a linear combination of any existing variables, allowing for multi-objective optimization. This is done using the `setObjective` (*coeffs*, *dir*) function call, which accepts a mapping *coeffs* of variable names to their coefficients. The binary input *dir* indicates whether the objective should be minimized or maximized.

B Additional applications

New elastic scaling capabilities: Here, we show SOL can be used for novel SDN applications. Specifically, we consider an elastic NFV setting [14] that places middleboxes in the network and allocates capacities in response to observed demand. There could be additional constraints, such as the total number of such VM locations. As a simple objective, we consider an upper bound on the number of nodes used while still load balancing across virtual middlebox instances. We can easily add other objectives such as minimizing number of VMs. For brevity we highlight only the key parts of building such a novel application.

```
1 predicate = hasMboxPredicate
2 opt.addBinaryVariables(pptc, topo, [’path’, ’node’])
3 opt.addNodeCapacityConstraint(pptc, ’cpu’, {node: ’TBA’
      for node in topo.nodes()}, lambda node, tc, path,
      resource: tc.volFlows * tc.cpuCost)
4 opt.addRequireSomeNodesConstraint(pptc)
5 opt.addPathDisableConstraint(pptc)
6 opt.addBudgetConstraint(topo, lambda node: 1, topo.
      getNumNodes()/2)
7 opt.setPredefinedObjective(’minmaxnodeload’, resource=’
      cpu’)
```

First, we define a valid path to be one that goes though a middlebox; SOL provides a predicate for that (line 1). The main difference here is the definition of capacities with the `TBA` value on line 3; this indicates that our optimization must allocate the capacities to the nodes. (SOL ensures that disabled nodes have 0 capacity allocated.) Thus we require at least one enabled node per path (lines 4, 5), limit the number of enabled nodes (line 6), and set the objective (line 7).

```
def _MaxMinFairness_MCF(topology, pptc, unsaturated,
     saturated, allocation, linkCaps):
    opt = getOptimization()
    opt.addDecisionVariables(pptc)
    # setup flow constraints
    opt.addAllocateFlowConstraint({tc: pptc[tc] for tc in
         unstaturated})
    for i in saturated:
        opt.addAllocateFlowConstraint({tc: pptc[tc] for
             tc in pptc[i]}, allocation[i])
    # setup link capacities:
    def linkcapfunc(link, tc, path, resource):
        return tc.volBytes
    opt.addLinkCapacityConstraint(pptc, ’bandwidth’,
         linkCaps, linkcapfunc)
    opt.setPredefinedObjective("maxallflow")
    opt.solve()
    return opt

def iterateMaxMinFairness(topology, pptc, linkCaps):
    # Setup saturated and unsaturated commodities
    saturated = defaultdict(lambda: [])
    unsaturated = set(pptc.keys())
    paths = defaultdict(lambda: [])

    t = []  # allocation values per each iteration
    i = 0  # iteration index
    while unsaturated:
        # Run slightly modified multi-commodity flow
        opt = _MaxMinFairness_MCF(topology, pptc,
             unsaturated, saturated, t, linkCaps)
        if not opt.isSolved():
            raise FormulationException(’No solution’)
        alloc = opt.getSolvedObjective()
        t.append(alloc)
        # Check if commodity is saturated, if so move it
             to saturated list
        for tc in list(unsaturated):
            # NOTE: this is an inefficient non-blocking
                 test, based on dual variables
            # More efficient methods are available
            dual = opt.getDualValue(opt.al(tc))
            if dual > 0:
                unsaturated.remove(tc)
                saturated[i].append(tc)
                paths[tc] = opt.getPathFractions()[tc]
        i += 1
    return paths
```

Figure 18: ***Python*** ***code for Max-min fairness optimization***

Complex multi-part optimizations: We also show how one can model more complex optimizations, using SOL as a primitive to express certain blocks of the optimization. Fig. 18 provides code for solving a max-min fairness problem. It relies on expressing intermediate multi-commodity flow problems using SOL (see function `_MaxMinFairness_MCF`) and writing a small iterative algorithm (see function `iterateMaxMinFairness`) for arriving at the optimal solution. We model our code after the algorithm suggested by Pióro et al. [38], however there are more recent and efficient proposals that can also be expressed in SOL (e.g., [3, 9]).

Paving the Way for NFV:
Simplifying Middlebox Modifications using StateAlyzr

Junaid Khalid, Aaron Gember-Jacobson, Roney Michael,
Anubhavnidhi Abhashkumar, Aditya Akella
University of Wisconsin-Madison

Abstract

Important Network Functions Virtualization (NFV) scenarios such as ensuring middlebox fault tolerance or elasticity require redistribution of internal middlebox state. While many useful frameworks exist today for migrating/cloning internal state, they require modifications to middlebox code to identify needed state. This process is tedious and manual, hindering the adoption of such frameworks. We present a framework-independent system, StateAlyzr, that embodies novel algorithms adapted from program analysis to provably and automatically identify all state that must be migrated/cloned to ensure consistent middlebox output in the face of redistribution. We find that StateAlyzr reduces man-hours required for code modification by nearly 20×. We apply StateAlyzr to four open source middleboxes and find its algorithms to be highly precise. We find that a large amount of, but not all, live state matters toward packet processing in these middleboxes. StateAlyzr's algorithms can reduce the amount of state that needs redistribution by 600-8000× compared to naive schemes.

1 Introduction

Network functions virtualization (NFV) promises to offer networks great flexibility in handling middlebox load spikes and failures by helping spin up new virtual instances and dynamically redistributing traffic among instances. Central to realizing the benefits of such elasticity and fault tolerance is the ability to handle *internal middlebox state* during traffic redistribution. Because middlebox state is dynamic (it can be updated for each incoming packet) and critical (its current value determines middlebox actions), the relevant internal state must be made available when traffic is rerouted to a different middlebox instance [16, 26, 30].

Recognizing this, and given the high-overhead and poor efficiency of existing approaches for replicating and sharing application state [16, 24, 26], researchers have developed several exciting frameworks for transferring, cloning, or sharing live middlebox state across instances, e.g., OpenNF [16], FTMB [30], Split/Merge [26], Pico Replication [24], and StatelessNF [20].

However, for middleboxes to work with these frameworks, middlebox developers must modify, or at least annotate, their code to perform custom state allocation, track updates to state, and (de)serialize state objects. The central contribution of this paper is a novel, framework-independent system that greatly reduces the effort involved in making such modifications.

Three factors make such modifications difficult today: (i) middlebox software is extremely complex, and the logic to update/create different pieces of state can be intricate; (ii) there may be 10s-100s of object types that correspond to state that needs explicit handling; and (iii) middleboxes are extremely diverse. Factors i and ii make it difficult to reason about the completeness or correctness of manual modifications. And, iii means manual techniques that apply to one middlebox may not extend to another. Our own experience in modifying middleboxes to work with OpenNF [16] underscores these problems. Making even a simple monitoring appliance (PRADS [6], with 10K LOC) OpenNF-compliant took over 120 man-hours. We had to iterate over multiple code changes and corresponding unit tests to ascertain completeness of our modifications; moreover, the process we used for modifying this middlebox could not be easily adapted to other more complex ones!

These difficulties significantly raise the bar for the adoption of these otherwise immensely useful state handling frameworks. To reduce manual effort and ease adoption, we develop StateAlyzr, a system that relies on *data and control-flow analysis* to automate identification of state objects that need explicit handling. Using StateAlyzr's output, developers can easily make framework-compliant changes to arbitrary middleboxes, e.g., identify which state to allocate using custom libraries for [20, 24, 26], determine where to track updates to state [16, 26, 30], (de)serialize relevant state objects for transfer/-cloning [16], and merge externally provided state with internal structures [16, 24]. In practice we find StateAlyzr to be highly effective. For example, leveraging StateAlyzr to make PRADS OpenNF-compliant took under 6 man-hours of work.

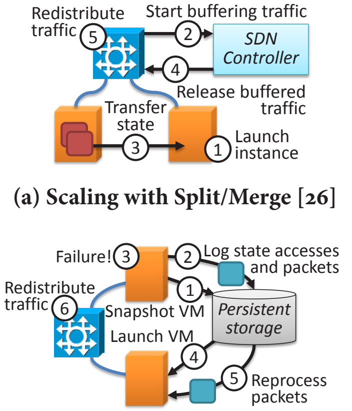
Importantly, transferring/cloning state objects identified with StateAlyzr is provably *sound* and *precise*. The former means that the aggregate output of a collection of instances following redistribution is equivalent to the output that would have been produced had redistribution not occurred. The latter means that StateAlyzr identifies minimal state to transfer so as to ensure that redistribution offers good performance and incurs low overhead.

However, achieving high precision without compromising soundness is challenging. Key attributes of middlebox code contribute to this: e.g., numerous data structures and procedures, large callgraphs, heavy use of (multi-level) pointers, and indirect calls to packet processing routines that modify state (See Table 2).

To overcome these challenges, StateAlyzr cleverly adapts program analysis techniques, such as slicing [18, 33] and pointer analysis [9, 31], to typical middlebox code structure and design patterns, contributing new algorithms for detailed classification of middlebox state. These algorithms can automatically identify: (*i*) variables corresponding to state objects that pertain to individual or groups of flows, (*ii*) the subset of these that correspond to state objects that can be updated by an arbitrary incoming packet at runtime, (*iii*) the flow space corresponding to a state object, (*iv*) middlebox I/O actions that are impacted by each state object, and (*v*) objects updated at runtime by an incoming packet.

To evaluate StateAlyzr, we both prove that our algorithms are sound (Appendix B) and use experiments to demonstrate precision and the resultant impact on the efficiency of state transfer/cloning. We run StateAlyzr on four open source middleboxes—Passive Real-time Asset Detection System (PRADS) [6], HAProxy load balancer [2], Snort Intrusion Detection System [7], and OpenVPN gateway [5]—and find:

- StateAlyzr's algorithms improve precision significantly: whereas the middleboxes have 1500-18k variables, only 29-131 correspond to state that needs explicit handling, and 10-148 are updateable at run time. By automatically identifying updateable state, StateAlyzr allows developers to focus on the necessary subset of variables among the many present. StateAlyzr can be imprecise: 18% of the updateable variables are mis-labeled (they are in fact read-only), but the information StateAlyzr provides allows developers to ignore processing these variables.
- Using StateAlyzr output, we modified PRADS and Snort to support fault tolerance using OpenNF [16]. We find that StateAlyzr reduces the manual effort needed. We could modify Snort (our most complex middlebox) and PRADS in 90 and 6 man-hours, respectively. Further, by helping track which flowspace an incoming packet belongs to, and which state objects it had updated, StateAlyzr reduces unneeded runtime state transfers between the primary and backup instances of PRADS and Snort by 600× and 8000× respectively compared to naive approaches.
- StateAlyzr can process middlebox code in a reasonable amount of time. Finally, it helped us identify important variables that we missed in our earlier modifications to PRADS, underscoring its usefulness.



(a) Scaling with Split/Merge [26]

(b) Failure recovery with FTMB [30]

Figure 1: Scaling and failure recovery process with recently state management frameworks

2 Motivation

A central goal of NFV is to create more scalable and fault tolerant middlebox deployments, where middleboxes *automatically scale* themselves in accordance with network load and *automatically heal* themselves when software, hardware, or link failures occur [4]. Scaling, and possibly fault tolerance, requires launching middlebox instances on demand. Both require redistributing network traffic among instances, as shown in Figure 1.

2.1 Need for Handling State

Middlebox scaling and failure recovery should be transparent to end-users and applications. Key to ensuring this is maintaining *output equivalence*: for any input traffic stream, the aggregate output of a dynamic set of middlebox instances should be equivalent to the output produced by a single, monolithic, always-available instance that processes the entire input [26]. The output may include network traffic and middlebox logs.

As shown in prior works [16, 26, 30], achieving output equivalence is hard because middleboxes are *stateful*. Every packet the middlebox receives may trigger updates to multiple pieces of internal state, and middlebox output is highly dependent on the current state. Thus, malfunctions can occur when traffic is rerouted to a middlebox instance without *the relevant internal state being made available at the instance*. Approaches like naively rerouting newly arriving flows or forcibly rerouting flows with pertinent state can violate output equivalence. The reader is referred to [16, 24] for a more formal treatment of the need to handle internal state.

2.2 Approaches for Handling State

Traditional approaches for replicating and sharing application state are resource intensive and slow [16, 24, 26]. Thus, researchers have introduced fast and efficient frameworks that transfer, clone, or share live internal middlebox state across instances. Examples include:

		Required Modifications			
		State	State	Serial-	Merge
Framework	Provides	Alloc.	Access	ization	State
Split/Merge [26]	Elasticity	✓	✓		✓
Pico Rep. [24]	Fault tol.	✓	✓		
OpenNF [16]	Both			✓	✓
FTMB [30]	Fault tol.		✓		
StatelessNF [20]	Both	✓	✓		

Table 1: Middlebox modifications in different frameworks

Split/Merge [26] and *StatelessNF* [20] that focus on elasticity; *Pico replication* [24] and *FTMB* [30] that focus on fault tolerance; and *OpenNF* [16] that applies to both. Unfortunately, these frameworks require detailed modifications to middlebox code to handle state (see Table 1):

- Split/Merge [26] and Pico Replication [24] require middleboxes to allocate and access all per- and cross-flow state—i.e., state that supports the processing of multiple packets within and across flows, respectively—through a specialized shared library, instead of using system-provided functions (e.g., malloc). This allows the frameworks to transfer and replicate middlebox state without serializing or updating middlebox-internal structures.
- OpenNF [16] requires middleboxes to identify and serialize per- and cross-flow state objects pertaining to a particular flowspace, as well as deserialize and integrate objects received from other middlebox instances. This allows OpenNF to transfer and copy flow-related state between middlebox instances.
- FTMB [30] requires middleboxes to log: (*i*) accesses to cross-flow state, and (*ii*) invocations of non-deterministic functions (e.g., `gettimeofday`). The logs allow FTMB to deterministically reprocess packets on a different middlebox instance in case the current instance fails before an up-to-date snapshot of its state can be captured.
- StatelessNF [20] requires middleboxes to create, read, and update all state values from a central, RDMA (remote direct memory access) based key/-value store. This enables any middlebox instance to have access to any state, and hence any instance can safely process any packet.

Making the above modifications to middleboxes is difficult because middlebox code is complex. As shown in Table 2, several popular middleboxes have between 60K and 275K lines of code (LOC), dozens of different structures and classes, and, in some cases, complex event-based control flow. If a developer misses a change to some structure, class, or function, then output equivalence may be violated under certain input patterns, and a middlebox may fail in unexpected ways at run time. FTMB is the only system that aims to avoid such problems. It automatically modifies middleboxes using LLVM [3]. However, there are two problems: (i) developers must still manually specify which variables may contain/point-to cross-flow state; (ii) the tool is limited to Click-based middleboxes [21].

2.3 Simplifying Modification and its Requirements

Making the aforementioned changes to even simple middleboxes can take numerous man-hours as our own experience with OpenNF suggests. This is a serious barrier to adopting any of the previously mentioned systems.

A system that can automatically identify what state a middlebox creates, where the state is created, and how the state is used could be immensely helpful in reducing the man-hours. It can provide developers guidance on writing custom state allocation routines, and on adding appropriate state filtering, serialization, and merging functions. Thus, it would greatly lower the barriers to adopting the above frameworks.

Building such a system is challenging because of *soundness* and *precision* requirements. Soundness means that the system must not miss any types, storage locations, allocations, or uses of state required for output equivalence. A precise system identifies the minimal set of state that requires special handling to ensure state handling at runtime is fast and low-overhead.

2.4 Options

Well-known *program analysis* approaches can be applied to identify middlebox state and its characteristics.

Dynamic analysis. We could use dynamic taint analysis [29] to monitor which pieces of state are used and modified while a middlebox processes some sample input. Unfortunately, the sample inputs may not exercise all code paths, causing the analysis to miss some state. We also find that such monitoring can significantly slow middleboxes down (e.g., PRADS [6] and Snort IDS [7] are slowed down > 10×).

Static analysis. Alternatively, we could use symbolic execution [10] or data-/control-flow analysis [15, 18].[1]

Symbolic execution can be employed to explore all possible code paths by representing input and runtime state as a series of symbols rather than concrete values. We can then track the state used in each path. While this is sound, the complexity of most middleboxes (Table 2) makes it impossible to explore all execution paths in a tractable amount of time. For example, we symbolically executed PRADS—which has just 10K LOC—for 8 hours using S2E [10], and only 13% of PRAD's code was covered. The complexity worsens exponentially for middleboxes with larger codebases. Recent advances in symbolic execution of middleboxes [14] do not help as they over come state space explosion by abstracting away middlebox state, which is precisely what we aim to analyze.

[1] Abstract interpretation [12] is another candidate, but it suffers from the well known problem of incompleteness, i.e., it over-approximates the middlebox's processing and may not identify all relevant state.

Middlebox	LOC (C/C++)	Classes/ Structs	Event based?	Level of pointers	Number of procedures	Size of callgraph
Snort IDS [7]	275K	898	No	10	4617	3391
HAProxy load balancer [2]	63K	191*	No	8*	2560	1018
OpenVPN [5]	62K	194*	No	2*	2023	1184
PRADS asset detector [6]	10K	40	No	4	297	115
Bro IDS [23]	97K	1798	No	-	3034	-
Squid caching proxy [8]	166K	875	Yes	-	2133	-

*Shows the lower bound. It does not include the number of structs used by the libraries and kernel.

Table 2: Code complexity for popular middleboxes. Those above the line are analyzed in greater detail later.

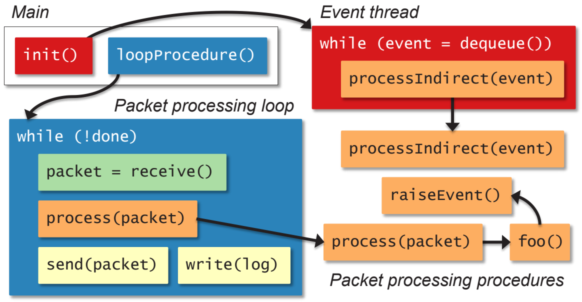


Figure 2: Logical structure of middlebox code

In this paper, we make clever use of *data-/control-flow analysis* to automatically evaluate how to handle middlebox state. Naively applying standard data-/control-flow analysis identifies all variables as pertaining to 'state that needs handling' (e.g., variables pertaining to per-packet state, read-only state, and state that falls outside the scope of a flowspace of interest); if developers modify a middlebox to specially handle all these variables, it can result in arbitrarily poor runtime performance during redistribution. We show how *middlebox code structure and design patterns* can be used to design *novel algorithms* that employ *static program analysis* techniques in a way that significantly improves *precision* without compromising *soundness*. Our approach is general and does not assume use of any particular state management framework.

3 Overview of StateAlyzr

Most middleboxes' code can be logically divided into three basic parts (Figure 2): initialization, packet receive loop, and packet processing. The initialization code runs when the middlebox starts. It reads and parses configuration input, loads supplementary modules or files, and opens log files. All of this can be done in the `main()` procedure, or in separate procedures called by `main`. The packet receive loop is responsible for reading a packet (or byte stream) from the kernel (via a socket) and passing it to the packet processing procedure(s). The latter analyzes, and potentially modifies, the packet. This procedure(s) reads/writes internal middlebox state to inform the processing of the current (and future) packet.

Our approach consists of three primary stages that leverage this structure. In each stage we further refine our characterization of a middlebox's state. The stages and their main challenges are described next:

1) Identify Per-/Cross-Flow State. In the first stage, we identify the storage location for all per- and cross-flow state created by the middlebox. The final output of this stage is a list of what we call *top-level variables* that contain or indirectly refer to such state.

Unlike state that is only used for processing the current packet, per-/cross-flow state influences other packets' processing. Consequently, the *lifetime* of this state extends beyond the processing a single packet. We leverage this property, along with knowledge of the relation between variable and value lifetimes, to first identify variables that may contain or refer to per-/cross-flow state.

We improve precision by considering which variables are actually *used* in packet processing code, thereby eliminating variables that contain or refer to state that is only used for middlebox initialization. We call the remaining variables "top-level". The main challenge here is dealing with indirect calls to packet processing in event-based middleboxes (Figure 2), which complicate the task of identifying all packet processing code. We develop an algorithm that adapts *forward program slicing* [18] to address this challenge (§4.1).

2) Identify Updateable State. The second stage further categorizes state based on whether it may be updated while a packet is processed. If state is read-only, we can avoid repeated cloning (in Pico Replication and OpenNF), avoid unnecessary logging of accesses (FTMB), and allow simultaneous access from multiple instances (StatelessNF); all of these will reduce the frameworks' overhead. We can trivially identify updateable state by looking for assignment statements in packet processing procedures. However, this strawman is complicated by heavy use of pointers in middlebox code which can be used to indirect state update. To address this challenge we show how to employ flow-, context-, and field-insensitive *pointer analysis* [9, 31] (§4.2).

3) Identify States' Flowspace Dimensions. Finally, the third stage determines a state's *flowspace*: a set of packet header fields (e.g. src_ip, dest_ip, src_port, dest_port & proto) that delineate the subset of traffic that relates to the state. Flowspace must be considered when modifying a middlebox to use custom allocation functions [24, 26] or filter state in preparation for export [16]. It is important to avoid the inclusion of irrelevant header fields and the exclusion of relevant fields in a state's flowspace, because it impacts runtime correctness and performance, respectively. To solve this problem we developed an algorithm that leverages common state access patterns in middleboxes to identify program points where we can apply *program chopping* [27] to determine relevant header fields (§4.3).

Soundness. In order for StateAlyzr to be sound it is necessary for these three stages to be sound. In Appendix B, we prove the soundness of our algorithms.

Assumptions about middlebox code. Our proofs are based on the assumption that middleboxes use standard API or system calls to read/write packets and hashtables or link-lists to store state. These assumption are not limitations of our analysis algorithms. Instead, they are made to ease the implementation of StateAlyzr. Our implementation can be extended to add additional packet read-/write methods or other data structures to store the state.

4 StateAlyzr Foundations

We now describe our novel algorithms for detailed state classification. To describe the algorithms, we use the example of a simple middlebox that blocks external hosts creating too many new connections (Figure 3).

4.1 Per-/Cross-Flow State

Our analysis begins by identifying the storage location for all relevant per- and cross-flow state created by the middlebox. This has two parts: (*i*) exhaustively identifying persistent variables to ensure soundness, and (*ii*) carefully limiting to top-level variables that contain or refer to per-/cross-flow values to ensure precision.

4.1.1 Identifying Persistent Variables

Because per-/cross-flow state necessarily influences two or more packets within/across flows, values corresponding to such state must be created during or prior to the processing of one packet, and be destroyed during or after the processing of a subsequent packet. Hence, the corresponding variables must be *persistent*, i.e., their values persist beyond a single iteration of the packet processing loop. In Figure 3, variables declared on lines 7 to 11 are persistent, whereas `curr` on line 61 is not. Our algorithm first identifies such variables.

Analysis Algorithm. We traverse a middlebox's code, as shown in Figure 4. The values of all global and static variables exist for the entire duration of the middlebox's execution, so these variables are always persistent. Variables local to the *loop-procedure*[2]—i.e., the procedure containing the packet processing loop—exist for the duration of this procedure, and hence the duration of the packet processing loop, so they are also persistent.

Local variables of procedures that precede the loop-procedure on the call stack are also persistent, because the procedures' stack frames last longer than the packet processing loop. However, these variables cannot be used within the packet processing loop, or a procedure called therein, because the variables are out of scope. Thus we exclude these from our list of persistent variables, improving precision.

```
1  struct host {
2    uint ip;
3    int count;
4    struct host *next;
5  }
6
7  pcap_t *intPcap, *extPcap;
8  int threshold;
9  char * queue[100];
10 int head = 0, tail = 0;
11 struct host *hosts = NULL;
12
13 int main(int argc, char **argv) {
14   pthread_t thread;
15   intPcap = pcap_create(argv[0]);
16   extPcap = pcap_create(argv[1]);
17   threshold = atoi(argv[2]);
18   pthread_create (&thread,(void*)&processPacket);
19 }
20
21 int loopProcedure() {
22   while(1) {
23     struct pcap_pkthdr pcapHdr;
24     char *pkt = pcap_next(extPcap, &pcapHdr);
25     ifFull_Wait();
26     enqueue(pkt);
27     if (entry->count < threshold)
28       pcap_inject(intPcap, pkt, pcapHdr->caplen);
29 } }
30
31 void enqueue(char* pkt){
32   head = (head + 1)%100;
33   queue[head] = pkt;
34 }
35
36 char* dequeue(){
37    int *index = &tail;
38    *index = (*index + 1)%100;
39    return queue[*index];
40 }
41
42 void processPacket(){
43   while(1){
44     ifEmpty_Wait();
45     char* pkt = dequeue();
46     struct ethhdr *ethHdr= (struct ethhdr)pkt;
47     struct iphdr *ipHdr= (struct iphdr*)(ethHdr+1);
48     struct tcphdr *tcpHdr= (struct tcphdr*)(ipHdr+1);
49     struct host *entry= lookup(ipHdr->saddr, hosts);
50     if (NULL == host){
51       struct host *new = malloc(sizeof(struct host));
52       new->ip = ipHdr->saddr;
53       new->next = hosts;
54       hosts = new;
55     }
56     if (tcpHdr->syn && !tcpHdr->ack)
57       entry->count++;
58 }  }
59
60 struct host *lookup(uint ip) {
61   struct host *curr = hosts;
62   while (curr != NULL) {
63     if (curr->ip == ip)
64       return curr;
65     curr = curr->next;
66 } }
```

Figure 3: Code for our running example.

The above analysis implicitly considers heap-allocated values by considering the values of global, static, and local variables, which can point to values on the heap. Values on the heap exist until they are explicitly freed (or the middlebox terminates), but their *usable lifetime* is lim-

[2] To automatically detect packet processing loops, we use the fact that middleboxes read packets using standard library/system functions.

```
Input: prog
Output: persistVars
persistVars = {}
persistVars = persistVars ∪ GlobalVarDecls(prog)
foreach proc in Procedures(prog) do
    persistVars = persistVars ∪ StaticVarDecls(proc)
persistVars = persistVars ∪ LocalVarDecls(loopProc)
persistVars = persistVars ∪ FormalParams(loopProc)
```

Figure 4: Identifying persistent variables

ited to the time frame in which they are reachable from a variable's value.[3] Therefore, we can conclude that a heap-allocated value's persistence is predicated on the persistence of a variable identified by our algorithm.

4.1.2 Limiting to Top-level Variables

The above algorithm identifies a superset of variables that may be bound, or point, to per-/cross-flow state. It includes variables bound to state used in initialization for loading/processing configuration/signature files: e.g., variables `intPcap` and `extPcap` in Figure 3. Such variables don't need handling during traffic redistribution; they can simply be copied when an instance is launched. To eliminate such variables and improve precision, the key insight we leverage is that, by definition, per-/cross-flow state is *used* in some way during packet processing. However, identifying all such variables is non-trivial, and missing variables impact analysis soundness.

```
   Input: prog, persistVars
   Output: pktProcs, percrossflowVars
 1 pktProcs = {}
 2 sdg = SystemDependenceGraph(prog)
 3 foreach stmt in Statements(loopProc) do
   //Statements() returns all statements in a procedure
 4     if stmt calls PKT_RECV_FUNC then
 5         slice = ForwardSlice(sdg, stmt, stmt.LHS)
 6         pktProcs = pktProcs ∪ Procedures(slice)
           //Procedures() returns all procedures in a slice
 7 percrossflowVars = {}
 8 foreach proc in pktProcs do
 9     foreach stmt in Statements(proc) do
10         foreach var in Vars(stmt) do
           //Vars() returns all variables used in a statement
11             if var in persistVars then
12                 percrossflowVars = percrossflowVars ∪ {var}
```

Figure 5: Identifying per-/cross-flow variables

Identifying Packet Processing Procedures. Figure 5 shows our algorithm for identifying top-level variables that contain or refer to per-/cross-flow values. The first half of the algorithm (lines 1–6) focuses on identifying packet processing code. Obviously any code contained in the packet processing loop is used for processing packets, but, crucially, the code of procedures (indirectly) called from within the loop is also packet processing code.

We considered a strawman approach of using call graphs to identify packet processing procedure. A call graph is constructed by starting at each procedure call within the packet processing loop, and classifying each appearing procedure as a packet processing procedure. However, this analysis does not capture packet processing procedures that are called indirectly. The Squid proxy, e.g., does initial processing of the received packet, then enqueues an event to trigger further processing through later calls to additional procedures. Hence the analysis may incorrectly eliminate some legitimate per-/cross-flow state which is used in such procedures.

Thus, we need an approach that exhaustively considers the dependencies between the receipt of a packet and both direct and indirect invocations of packet processing procedures. Below, we show how *system dependence graphs* [15] and *program slicing* [18] can be used for this.

A *system dependence graph* (SDG) consists of multiple program dependence graphs (PDGs) — one for each procedure. Each PDG contains vertices for each statement along with their data and control dependency edges. A *data dependence* edge is created between statements *p* and *q* if there is an execution path between them, and *p* may update the value of some variable that *q* reads. A *control dependence* edge is created if *p* is a conditional statement, and whether or not *q* executes depends on *p*. A snippet of the control and data edges for our example in Figure 3 is in Figure 6.

Whereas control edges capture direct invocations of packet processing, we can rely on data edges to capture indirect procedure calls. For example, the dashed yellow lines in Figure 6 fail to capture invocation of the `processPacket` procedure on bottom right (because there is no control edge from the while loop or any of its subsequent procedures to `processPacket`). In contrast, we can follow the data edges, the dashed red line, to track such calls.

Given a middlebox's SDG, we compute a *forward program slice* from a packet receive function call for the variable which stores the received packet. A forward slice contains the set of *statements that are affected* by the value of a variable starting from a specific point in the program [18]. Most middleboxes use standard library/system functions to receive packets—e.g., `pcap_next`, or `recv`—so we can easily identify these calls and the variable pointing to the received packet. We consider any procedure appearing in the computed slice to be a packet processing procedure. For middleboxes which invoke packet receive functions at multiple points, we compute forward slices from every call site and take the union of the procedures appearing in all such slices.

Values Used in Packet Processing Procedures. The second half of our algorithm (Figure 5, lines 7–12) focuses on identifying persistent values that are used within some

[3] A heap value whose lifetime is longer than its usable lifetime is a memory leak.

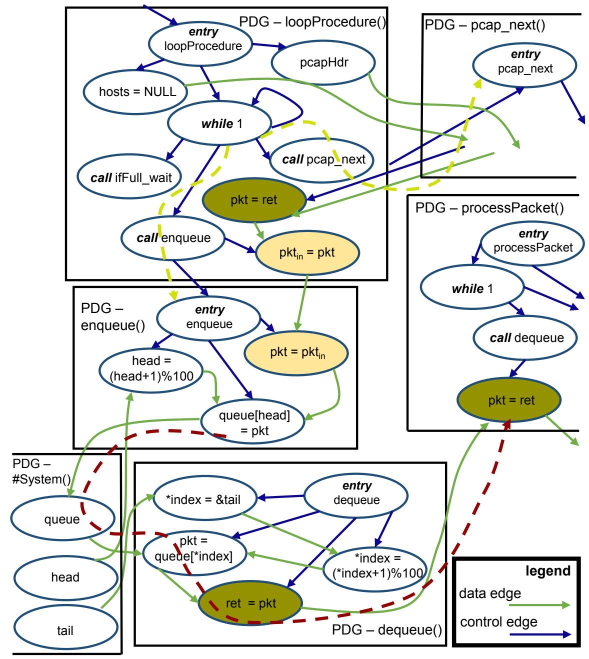


Figure 6: Snippet of System dependence graph (SDG) for the code in Figure 3; green edges indicate data dependencies and blue edges indicate control dependencies; light yellow nodes represent formal and actual parameters, while dark yellow nodes represent return values.

packet processing procedure. We analyze each statement in the packet processing procedures. If the statement contains a persistent variable, then we mark that persistent variable as a top-level variable.

4.2 Updateable State

Next, we delineate *updateable* top-level variables from *read only* variables to further improve precision. In Figure 3, variable `head`, `tail`, `hosts` and `queue` are updateable, whereas `threshold` is not. Because state is updated through assignment statements, one strawman choice here is to statically identify top-level variables on the left-hand-side (LHS) of assignment statements. In Figure 3, this identifies `head`, `hosts` and `queue`.

However, this falls short due to *aliasing*, where multiple variables are bound to the same storage location due to the use of pointers [11]. Aliasing allows a value reachable from a top-level variable to be updated through the use of a different variable. Thus our strawman can mis-label top-level variables as read-only, compromising soundness. For example, `tail` is mislabeled in Figure 3, because it never appears on the LHS of assignment statements. But on line 38 `index` is updated which points to `tail`.

Analysis Algorithm. We develop an algorithm to identify *updateable top-level variable* (Figure 7). Since we are concerned with variables whose (referenced) values are updated during packet processing, we analyze each assignment statement contained in the packet processing procedures identified in the first stage of our analysis (§4.1.2). If the assignment statement's LHS contains a top-level variable, then we mark the variable as updateable (similar to our strawman). Otherwise, we compute the *points-to* set for the variable on the LHS and compare this with the set of updateable top-level variables and their points-to sets. A variable's points-to set contains all variables whose associated storage locations are reachable from the variable. To compute this set, we employ flow-, context-, and field-insensitive pointer analysis [9]. If the points-to set of the variable on the LHS contains a top-level variable, or has a non-null intersection with the points-to set of a top-level variable, then we mark the top-level variable as updateable.

```
Input: $pktProcs$, $percrossflowVars$
Output: $updateableVars$
$percrossflowVars = \{\}$
foreach $proc$ in $pktProcs$ do
    foreach $stmt$ in AssignmentStmts($proc$) do
    //AssignmentStmts() returns all assignment
    statements in a procedure
        foreach $var$ in $percrossflowVars$ do
            if $stmt$.LHS == $var$
                or $var$ in PointsTo($stmt$.LHS)
                or PointsTo($var$) ∩ PointsTo($stmt$.LHS) ≠ ∅
            then
                $updateableVars = updateableVars \cup \{var\}$
```

Figure 7: Identifying updateable variables

Due to limitations of pointer analysis, our algorithm may still mark read-only top-level variables as updateable. E.g., field insensitive pointer analysis can mark a top-level struct variable as updateable even if just one of its subfields is updateable.

4.3 State Flowspaces

Finally, we identify the packet header fields that define the flowspace associated with the values of each top-level variable. Identifying too fine-grained of a flowspace for a value—i.e., more header fields than those that actually define the flowspace—is unsound; such an error will cause a middlebox to incorrectly filter out the value when it is requested by a middlebox state management framework [16, 20, 24, 26]. Contrarily, assuming an overly permissive flowspace (e.g., the entire flowspace) for a value hurts precision.

To identify flowspaces, we leverage common middlebox design patterns in updating or accessing state. Middleboxes typically use simple data structures (e.g., a hash table or linked list) to organize state of the same type for different network entities (connections, applications, subnets, URLs, etc.). When processing a packet, a middlebox uses header fields[4] to lookup the entry in the

[4] In cases where keys are not based on the packet header fields e.g. URL, a middlebox usually keeps another data structure to maintain the

data structure that contains a reference to the values that should be read/updated for this packet. In the case of a hash table, the middlebox computes an *index* from the packet header fields to identify the entry pointing to the relevant values. For a linked list, the middlebox *iterates* over entries in the data structure and compares packet header fields against the values pointed to by the entry.

```
   Input: pktProcs, percrossflowVars
   Output: chop, flowspace
1  $keyedVars = \{\}$
2  foreach var in percrossflowVars do
3      if Type(var) == pointer
         or Type(var) == struct then
4          $keyedVars = keyedVars \cup \{keyedVars\}$
5  foreach proc in pktProcs do
6      foreach loopStmt in LoopStmts(proc) do
7          $condVars = \{\}$
8          foreach var in Vars(loopStmt.condition) do
9              if var in keyedVars
                 or $\texttt{PointsTo}(var) \cap keyedVars \neq \emptyset$ then
10                 for condStmt in
                   ConditionalStmts(loopStmt.body) do
11                     for condVar in Vars(condStmt) do
12                         if $condVar \neq var$ then
13                             $condVars = condVars \cup \{condVar\}$
14         chop = Chop(sdg,pktVar,condVars)
15         flowspace = ExtractFlowspace(chop)
```

Figure 8: Identifying packet header fields that define a per-/cross-flow variable's associated flowspace

Algorithm. We leverage the above design patterns in our algorithm shown in Figure 8. In the first step (lines 2-4), if the top-level variable is a struct or a pointer, we mark it as a possible candidate for having a flowspace associated with it. This filters out all the top-level variables which cannot represent more than one entry; e.g., variables `head` and `tail` in Figure 3.

We assume that middleboxes use hash tables or linked lists to organize their values,[5] and that these data structures are accessed using:

square brackets, e.g.

```
entry = table[index];
```

pointer arithmetic, e.g.

```
entry = head + offset;
```

or iteration[6], e.g.

```
while(entry->next!=null){entry=entry->next;}
for(i=0; i<list.length; i++) {...}
```

The second step is thus to identify all statements like these where a top-level variable marked above is on the right-hand-side (RHS) of the statement (square brackets or pointer arithmetic scenario) or in the conditional expression (iteration scenario).

When square brackets or pointer arithmetic are used, we compute a *chop* between the variables in the access statement and the variable containing the packet returned by the packet receive procedure. A chop between a set of variables U at program point p and a set of variables V at program point q is the set of statements that (i) may be affected by the value of variables in U at point p, and (ii) may affect the values of variables in V at point q. Thus, the chop we compute above is a snippet of executable code which takes a packet as input and outputs the index or offset required to extract the value from the hashtable.

In a similar fashion, when iteration is used, we identify all conditional statements in the body of the loop. We compute a chop between the packet returned by the packet receive procedure and the set of all the variables in the conditional expression which do not point to any of the top-level variables; in our example (Figure 3), the chop starts at line 24 and terminates at line 63. We output the resulting chops, which collectively contain all conditional statements that are required to lookup a value in a linked list data structure based on a flow space definition. Assuming that the middlebox accesses packet fields using standard system-provided structs (e.g., `struct ip` as defined in `netinet/ip.h`), we conduct simple string matching on the code snippets to produce a list of packet header fields that define a state's flowspace.

5 Enhancements

Data and control flow analysis can help improve precision, but they have some limitations in that they cannot guarantee that exactly the relevant state and nothing else has been identified. In particular, static analysis cannot differentiate between multiple memory regions that are allocated through separate invocations of malloc from the same call site. Therefore, we cannot statically determine if only a subset of these memory regions have been updated after processing a set of packets. To overcome potential efficiency loss due to such limitations, we can employ custom algorithms that boost precision in specific settings. We present two candidates below.

5.1 Output-Impacting State

In addition to the three main code blocks (Figure 2), middleboxes may optionally have packet and log output functions. These pass a packet to the kernel for forwarding and record the middlebox's observations and actions in a log file, respectively. These functions are usually called from within the packet processing procedure(s).

In some cases, operators may desire output equivalence only for specific types of output. For example, an operator may want to ensure client connections are not broken when a NAT fails—i.e., packet output should be equivalent—but may not care if the log of NAT'd connec-

mapping between such keys and packet header fields

[5] Our approach can easily be extended to other data structures.

[6] Middleboxes may also use recursion, but we have not found this access pattern in the middleboxes we study, so we do not consider it in our algorithm.

tions is accurate. In such cases, internal state that only impacts non-essential forms of output does not need special handling during redistribution and can be ignored.

To aid such optimizations, we develop an algorithm to identify the type of output that updateable state affects. We use two key insights. First, middleboxes typically use *standard libraries and system calls* to produce packet and log output: either PCAP (e.g. pcap_dump) or socket (e.g. send) functions for the former, and regular I/O functions (e.g. write) for the latter.[7] Second, the output produced by these functions can only be impacted by a *handful of parameters* passed to these functions. Thus, we focus on the call sites of these functions, and their parameters.

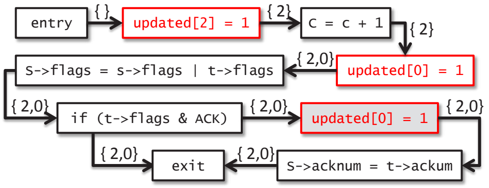
Algorithm. We use *program slicing* [18] to identify the dependencies between a specific type of output and updateable variables. We sketch the algorithm and relegate details to Appendix A. We first identify the call sites of packet or log output functions by checking each statement in each packet processing procedure (§4.1.2). Then we use the SDG produced in the first stage of our analysis (Figure 5) to compute a *backward slice* from each call site. Such a slice contains the set of statements that affect (*i*) whether the procedure call is executed, and (*ii*) the value of the variables used in the procedure call, such as the parameters passed to the output function. We examine each statement in a backward slice to determine whether it contains an updateable per-/cross-flow variable. Such variables are marked as impacting packet (or log) output.

5.2 Tracking Runtime Updates

Developers aiming to design fault-tolerant middleboxes can use the algorithms in §4 and §5.1 to efficiently clone state to backup instances. For example, if traffic will be distributed among multiple instances in the case of failure, then only state whose flowspace overlaps with that assigned to a specific instance needs to be cloned to that instance. However, the potential performance gains from these optimizations may be limited due to constraints imposed by data/control-flow analysis. For example, our analysis can only identify whether a persistent variable's value *may* be updated during the middlebox's execution. If we can determine at runtime exactly which values are updated, and when, then we can further improve the efficiency of state cloning and speed up failover.

```
1  struct conn tbl[1000]; // Assigned id 0
2  int count; // Assigned id 1
3  int tcpcnt; // Assigned id 2
4  char updated[3];
5  void main() {
6    while(1) {
7      char *pkt = recv();
8      updated[1] = 1;
9      count = count + 1;
10     struct *iphdr i = getIpHdr(pkt);
11     if (i->protocol == TCP) {
12       hdl(&tcpcnt, &tbl[hash(pkt)], getTcpHdr(pkt));
13 } } }
14 void hdl(int *c,struct conn *s,struct tcphdr *t) {
15   updated[2] = 1;
16   c = c + 1;
17   updated[0] = 1;
18   s->flags = s->flags | t->flags;
19   if (t->flags & ACK)
20     updated[0] = 1; // Pruned
21     s->acknum = t->acknum;
22 } }
```

(a) Example middlebox code instrumented for update tracking at run time; statements in red are inserted based on our analysis



(b) Annotated control flow graph used for pruning redundant updated-bit-setting (shaded) statements

Figure 9: Implementing update tracking at run time

To achieve higher precision, we must use (simple) run time monitoring. For example, we can track, at run time, whether part of an object is updated during packet processing. To implement this monitoring, we must modify the middlebox to set an "updated bit" whenever a value reachable from a top-level variable is updated during packet processing. Figure 9a shows such modifications, in red, for a simple middlebox. We create a unique updated bit for each top-level variable—there are three such variables in the example—and we set the appropriate bit before any statement that updates a value that may be reachable from the corresponding variable.

We use the same analysis discussed in §4.2 to determine where to insert statements to set updated bits. For any statement where a top-level variable is updated, we insert a statement—just prior to the assignment statement—that sets the appropriate updated bit.

However, this approach can add a lot more code than needed: if one assignment statement *always* executes before another, and they *always* update the same value, then we only need to set the updated bit before the first assignment statement. For example, line 21 in Figure 9a updates the same compound value as line 18, so the code on line 20 is redundant.

We use a straightforward control flow analysis to prune unneeded updated-bit-setting statements. First, we construct a control flow graph (CFG) for each modified packet processing procedure. Next, we perform a depth-first traversal of each CFG, tracking the set of updated bits that have been set along the path; as we traverse each edge, we label it with the current set of updated bits. Figure 9b shows this annotated CFG for the handleTcp procedure shown in lines 14-22 of Figure 9a. Lastly, for each updated-bit-setting statement in a procedure's CFG, we check whether the bit being set is included in the label for

[7] Our approach can be easily extended to consider non-standard output functions.

every incoming edge. If this is true, then we prune the statement; e.g., we prune line 20 in Figure 9a.

6 Implementation

We implement StateAlyzr using CodeSurfer [1] which has built-in support for constructing CFGs, performing flow- and context-insensitive pointer analysis, constructing PDGs/SDGs, and computing forward/backward slices and chops for C/C++ code. CodeSufer uses proven sound algorithms to implement these static analysis techniques. We use CodeSurfer's Scheme API to access output from these analyses in our algorithms. We applied StateAlyzr to four middleboxes: PRADS asset monitoring [6] and Snort Intrusion Detection System [7], HAproxy load balancer [2], and OpenVPN gateway [5].

Fault Tolerance. We use the output from StateAlyzr to add fault tolerance to PRADS and Snort, both off-path middleboxes. We added code to both to export/import internal state (to a standby). We used the output of our first two analysis phases (§4.1 and §4.2) to know which top level variables' values we need to export, and where in a hot-standby we should store them. We used the output of our third analysis phase (§4.3) as the basis for code that looks up per-/cross-flow state values. This code takes a flowspace as input and returns an array of serialized values. We use OpenNF [16] to transfer serialized values to a hot-standby. Similarly, import code deserializes the state and stores it in the appropriate location. We also implemented both enhancements discussed in §5.

7 Evaluation

We report on the outcomes of applying StateAlyzr to four middleboxes. We address the following questions:

- *Effectiveness*: Does StateAlyzr help with making modifications to today's middleboxes? How many top-level variables do these middleboxes maintain, relative to all variables? What relative fractions of these pertain to state that may need to be handled during redistribution? How precise is StateAlyzr?
- *Runtime efficiency and manual effort:* To what extent do StateAlyzr's mechanisms help improve the runtime efficiency of state redistribution? How much manual effort does it save?
- *Practical considerations*: Does StateAlyzr take prohibitively long to run (like symbolic execution; §2.4)? Is it sound in practice?

7.1 Effectiveness

In Table 3, we present a variety of key statistics derived for the four middleboxes using StateAlyzr. We use this to highlight StateAlyzr's ability to improve precision, thereby underscoring its usefulness for developers.

The complexity of middlebox code is underscored by the overall number of variables in Table 3, which can vary between 1500 and 18k, and other relevant code complexity metrics shown in Table 2. Thus, manually identifying state that needs handling, and optimizing its transfer, is extremely difficult.

Mbox	All	Persistent	Top -level	Update -able	pkt/log output impacting	require serial- ization
PRADS	1529	61	29	10	N.A. / 6	14
Snort	18393	507	333	148	N.A. /148	176
HAproxy	7876	272	176	115	101 / 109	59
OpenVPN	8704	156	131	106	97 / 102	8

Table 3: Variables and their properties

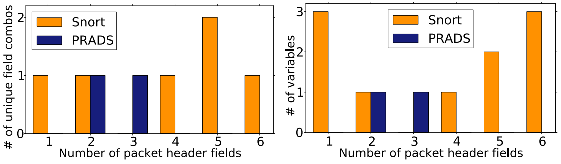


Figure 10: Flowspace dims. of keyed per-/cross-flow vars

We also note from Table 3 that StateAlyzr identifies 61-507 variables as persistent across the four middleboxes. A subset of these, 29-333, are top-level variables. Finally, 6-148 top-level variables are updateable; operators only have to deal with handling the values pertaining to these variables at run time. Snort is the most complex middlebox we analyze (≈275K lines of code) and has the largest number of top-level variables (333); the opposite is true for PRADS (10K LOC and 29 top-level variables).

The drastic reduction to the final number of updateable variables shows that naive approaches that attempt to transfer/clone values corresponding to *all* variables can be very inefficient at runtime. (We show this empirically in §7.2.) Even so, the number of updateable variables can be as high as 148, and attempting to manually identify them and argument code suitably can be very difficult. By automatically identifying them, StateAlyzr simplifies modifications; we provide further details in §7.2.

Finally, the reductions we observe in going from persistent variables to top-level variables (16-53% reduction) and further to updateable ones (19-65% reduction) show that our techniques in §4.1 and §4.2 offer useful improvements in precision.

In Figure 10, we characterize the flowspaces for the variables found in Snort and PRADS. From the left figure, we see that Snort maintains state objects that could be keyed by as many as 5 or 6 header fields; the maximum number of such fields for PRADS is 3. The figure on the right shows the number of variables that use a particular number of header fields as flowspace keys; for instance, in the case of Snort, 3 variables each are keyed on 1 and 6 fields. The total number of variables keyed on at least one key is 2 and 10 for Snort and PRADS, respectively (sum of the heights of the respective bars).

These numbers are significantly lower than the updateable variables we discovered for these middleboxes (6 and 148, respectively). Digging deeper into Snort (for example) we find that:

- 111 updateable variables pertain to all flows (i.e., a flowspace key of "*"). Of these, 59 variables are related to configurations and signatures, while 30 are function pointers (that point to different detection and processing plugins). These 89 variables can be updated from the command line at middlebox run time (when an operator provides new configurations and signatures, or new analysis plugins).
- 27 updateable variables—or 18%—are only used for processing a *single packet*; hence they don't correspond to per-/cross-flow state. This points to StateAlyzr's *imperfect precision*. These variables are global in scope and are used by different functions for processing a single incoming packet, which is why our analysis labels them as updateable. A developer can easily identify these variables and can either remove them from the list of updateable variables or modify code to make them local in scope.

7.2 Runtime efficiency and manual effort

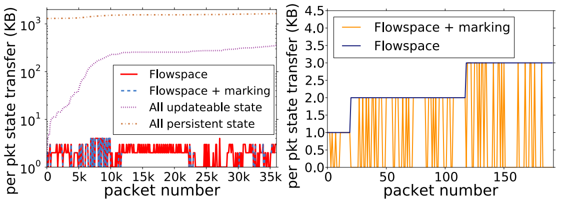
7.2.1 Fault Tolerant Middleboxes

Using fault tolerant PRADS/Snort versions (§5), we show that StateAlyzr helps significantly cut unneeded state transfers, improving state operation time/overhead.

Man-hours needed. Modifying PRADS based on StateAlyzr analysis took roughly 6 man-hours, down from over 120 man-hours when we originally modified PRADS for OpenNF (Two different persons made these modifications.). Modifying Snort, a much more complex middlebox, took 90 man-hours. In both cases, most of the time (> 90%) was spent in writing serialization code for the data structures identified by StateAlyzr (14 for PRADS and 176 for Snort; Table 3). Providing support for exporting/importing state objects according to OpenNF APIs took just 1 and 2 hours, respectively.

Runtime benefits. We consider a primary/hot standby setup, where the primary sends a copy of the state to the hot standby after processing each packet. We use a university-to-cloud packet trace [17] with around 700k packets for our trace-based evaluation of this setup. The primary instance processes the first half of the trace file until a random point, and the hot standby takes over after that. We consider three models for operating the hot standby which reflect progressive application of the different optimizations in §4 and §5: (i) the primary instance sends a copy of all the updateable states to the hot standby, (ii) the primary instance only sends the state which applies to the flowspace of the last processed packet, and (iii) in addition to considering the flowspace, we also consider which top level variables are marked as updated for the last processed packet.

Figure 11a shows the average case results for the amount of per packet data transferred between the primary and secondary instances for all three models for PRADS.



(a) (b)

Figure 11: (a) Per packet state transfer (b) Per packet state transfer for a single connection

Transferring state which only applies to the flowspace of the last processed packet, i.e., the second model, reduces the data transferred by 305× compared to transferring all per-/cross-flow state. Furthermore, we find that the third model, i.e., run time marking of updated state variables, further reduces the amount of data transferred by 2×, on average. This is because not all values are updated for every packet: the values pertaining to a specific connection are updated for every packet of that connection, but the values pertaining to a particular host and its services are only updated when processing certain packets. This behavior is illustrated in Figure 11b, which shows the size of the state transfer after processing each of the first 200 packets in a randomly selected flow.

We measured the increase in per packet processing time purely due to the code instrumentation needed to identify state updates for highly available PRADS. We observed an average increase of 0.04μsec, which is around 0.14% of the average per packet processing time for unmodified PRADS.

Figure 12 shows the corresponding results for Snort. Transferring just the updateable state results in a 8800× reduction in the amount of state transferred compared to transferring all per-/cross-flow state. This is because, a significant portion of the persistent state in Snort consists of configuration and signatures which are never updated during packet processing. Transferring state which only applies to a particular flowspace further reduces the data transfer by 2.75×. Unlike PRADS, the amount of state transfer in the second model remains constant for a particular flow because most of the state is created on the first few packets of a flow. Finally, runtime marking further reduces the amount of state transferred by 3.6×.

7.2.2 Packet/Log Output

Table 3 includes the number of variables that impact packet or log output. For on-path HAproxy (OpenVPN), 87% (91%) of updateable variables affect packet output; a slightly higher fraction impact log output. 95 (93) variables impact both outputs. A much smaller number impacts packet output but not log (6 and 4, respectively). Another handful impact logs but not packets (14 and 9); operators who are interested in just packet output consistency can ignore transferring the state pertaining to these variables, but the benefit will likely not be significant for

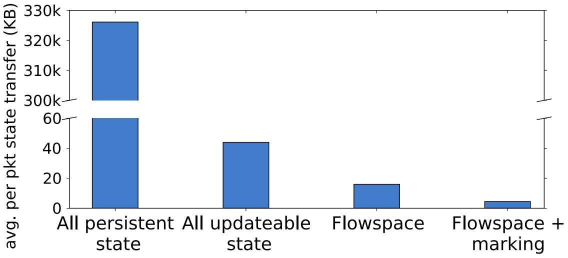


Figure 12: Per packet state transfer in Snort

these middleboxes given the low counts.

Being off-path, PRADS and Snort have no variables that impact packet output. For PRADS, 6 out of 10 updateable variables impact log output. StateAlyzr did find 4 other updateable variables—`tos`, `tstamp`, `in_pkt`, and `mtu`—but did not mark them as affecting packet output or log output. Upon manual code inspection we found that these values are updated as packets are processed, but they are never used; thus, these variables can be removed from PRADS without any impact on its output, pointing to another benefit of StateAlyzr—code clean-up.

7.3 Practicality

Table 4 shows the time and resources required to run our analysis. CodeSurfer computes data and control dependencies and points-to sets at compile time, so the middleboxes take longer than normal to compile. This phase is also memory intensive, as illustrated by peak memory usage. Snort, being complex, takes the longest to compile and analyze (≈20.5h). This is not a concern since StateAlyzr only needs to be run once, and it runs offline.

7.3.1 Empirically Verifying Soundness

Empirically showing soundness in practice is hard. Nevertheless, for the sake of completeness, we use two approaches to verify soundness of the modifications we make on the basis of StateAlyzr's outputs.

First, we use the experimental harness from §7.2. We compare logs at PRADS/Snort in the scenario where a single instance processes the complete trace file against concatenated logs of the primary and hot standby, using the trace and the three models as above. In all cases, there was no difference in the two sets of logs.

Next, we compare with manually making all changes. Recall that we had manually modified PRADS to make it OpenNF-compliant. We compared StateAlyzr's output for PRADS against the variables contained in the state transfer code we added during our prior modifications to PRADS. StateAlyzr found all variables we had considered in our prior modifications, and more. Specifically, we found that our prior modifications had *missed* an important compound value that contains a few counters along with configuration settings.

Mbox	Compile time	Analysis time	Memory
PRADS	0.2	0.25	0.3
Snort	1.5	19	6
HAproxy	0.25	6	6
OpenVPN	0.5	5	7.3

Table 4: Time (h) and memory usage (GB)

8 Other Related Work

Aside from the works discussed in §2 and §4 [9, 16, 18, 22, 24, 25, 26, 28, 31, 33] StateAlyzr is related to a few other efforts. Some prior studies have focused on transforming non-distributed applications into distributed applications [19, 32]. However, these works aim to run different parts of an application at different locations. We want all analysis steps performed by a middlebox instance to run at one location, but we want different instances to run on a different set of inputs without changing the collective output from all instances.

Dobrescu and Argyarki use symbolic execution to verify middlebox code satisfies crash-freedom, bounded-execution, and other safety properties [14]. They employ small, Click-based middleboxes [21] and abstract away accesses to middlebox state. In contrast, our analysis focuses on identifying state needed for correct middlebox operation and works with regular, popular middleboxes.

Lorenzo et al. [13] use similar static program analysis techniques to identify flowspace, but their identification is limited to just hashtables.

9 Summary

Our goal was to aid middlebox developers by identifying state objects that need explicit handling during redistribution operations. In comparison with today's manual and necessarily error-prone techniques, our program analysis based system, StateAlyzr, vastly simplifies this process, and ensures soundness and high precision. Key to StateAlyzr is novel state characterization algorithms that marry standard program analysis tools with middlebox structure and design patterns. StateAlyzr results in nearly 20× reduction in manual effort, and can automatically eliminate nearly 80% of variables in middlebox code for consideration during framework-specific modifications, resulting in dramatic performance and overhead improvements in state reallocation. Ultimately, we would like to fully automate the process of making middlebox code framework-compliant, thus fulfilling the promise of using NFV effectively for middlebox elasticity and fault tolerance. Our work addresses basic challenges in code analysis, a difficult problem on its own which is necessary to solve first.

Acknowledgments

We thank our shepherd, Mona Attariyan, and the anonymous reviewers for their insightful feedback. This work is supported in part by National Science Foundation (grants

CNS-1302041, CNS-1330308 and CNS-1345249) and the Wisconsin Institute on Software-Defined Datacenters of Madison.

References

[1] Codesurfer. `http://grammatech.com/research/technologies/codesurfer`.

[2] HAProxy: The reliable, high performance TCP/HTTP load balancer. `http://haproxy.1wt.eu/`.

[3] The LLVM compiler infrastructure. `http://llvm.org`.

[4] Network functions virtualisation – update white paper. `https://portal.etsi.org/nfv/nfv_white_paper2.pdf`.

[5] OpenVPN. http://openvpn.net.

[6] Passive Real-time Asset Detection System. `http://prads.projects.linpro.no`.

[7] Snort. `http://snort.org`.

[8] Squid. `http://squid-cache.org`.

[9] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[10] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[11] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, 1993.

[12] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT*, 1977.

[13] L. De Carli, R. Sommer, and S. Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1378–1390. ACM, 2014.

[14] M. Dobrescu and K. Argyarki. Software dataplane verification. In *NSDI*, 2014.

[15] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[16] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *SIGCOMM*, 2014.

[17] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 177–190. ACM, 2013.

[18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[19] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *OSDI*, 1999.

[20] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless network functions. In *HotMiddlebox*, 2015.

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18:263–297, 2000.

[22] Y. G. Park and B. Goldberg. Escape analysis on lists. In *PLDI*, 1992.

[23] V. Paxson. Bro: a system for detecting network intruders in real-time. In *USENIX Security (SSYM)*, 1998.

[24] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A high availability framework for middleboxes. In *SoCC*, 2013.

[25] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Escape capsule: Explicit state is robust and scalable. In *HotOS*, 2013.

[26] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System support for elastic execution in virtual middleboxes. In *NSDI*, 2013.

[27] T. Reps and G. Rosay. Precise interprocedural chopping. In *ACM SIGSOFT*, 1995.

[28] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *POPL*, 1988.

[29] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[30] J. Sherry, P. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Macciocco, M. Manesh, J. Martins, S. Ratnasamy, and L. R. S. Shenker. Rollback recovery for middleboxes. In *SIGCOMM*, 2015.

[31] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[32] E. Tilevich and Y. Smaragdakis. J-orchestra: Enhancing java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1):1:1–1:40, Aug. 2009.

[33] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, July 1984.

Appendix

A. Output-Impacting State - Algorithm

Figure 13 outlines the algorithm for identifying state that impacts packet/log output (from §5.1).

```
Input: sdg, updateableVars
Output: pktoutputVars, logoutputVars
pktoutputVars = {}
logoutputVars = {}
foreach proc in pktProcs do
  foreach stmt in Statements(proc) do
    if stmt calls PKT_OUTPUT_FUNC
      or stmt calls LOG_OUTPUT_FUNC then
      slice = BackwardSlice(sdg, stmt,
              Vars(stmt.RHS))
      foreach sliceStmt in Statements(slice) do
        foreach var in Vars(sliceStmt) do
          if var in updateableVars then
            if stmt calls PKT_OUTPUT_FUNC then
              pktoutputVars = pktoutputVars ∪ {var}
            else
              logoutputVars = logoutputVars ∪ {var}
```

Figure 13: Identifying output-impacting variables

B. Proofs of soundness

We now prove the soundness of our algorithms.

Identifying Per-/Cross-Flow State

Slicing [18] and pointer analysis [9] have already been proven sound.

Theorem 1. *If a middlebox uses standard packet receive functions, then our analysis identifies all packet processing procedures.*

Proof. For a procedure to perform packet processing: (i) there must be a packet to process, and (ii) the procedure must have access to the packet, or access to values derived from the packet. The former is true only after a packet receive function returns. The latter is true only if some variable in a procedure has a data dependency on the received packet. Therefore, a forward slice computed from a packet receive function over the variable containing (a pointer to) the packet will identify all packet processing procedures. □

Theorem 2. *If a value is per-/cross-flow state, then our analysis outputs a top-level variable containing this value, or containing a reference from which the value can be reached (through arbitrarily many dereferences).*

Proof. Assume no top-level variable is identified for a particular per-/cross-flow value. By the definition, a per-/cross-flow must (i) have a lifetime longer than the lifetime of any packet processing procedure, and (ii) be used within some packet processing procedure. For a value to be used within a packet processing procedure, it must be the value of, or be a value reachable from the value of, a variable that is in scope in that procedure. Only global variables and the procedure's local variables will be in scope.

Since we identify statements in packet processing procedures that use global variables, and points-to analysis is sound [9], our analysis must identify a global variable used to access/update the value; this contradicts our assumption.

This leaves the case where a local variable is used to access/update the value. When the procedure returns the variable's value will be destroyed. If the variable's value was the per-/cross-flow value, then the value will be destroyed and cannot have a lifetime beyond the packet processing procedure; this is a contradiction. If the variable's value was a reference through which the per-/cross-flow value could be reached, then this reference will be destroyed when the procedure returns. Assuming a value's lifetime ends when there are no longer any references to it, the only way for the per-/cross-flow value to have a lifetime beyond any packet processing procedure is for it be reached through another reference. The only such reference that can exist is through a top-level variable. Since points-to analysis is sound [9] this variable would have been identified, which contradicts our assumption. □

Identifying Updateable State

Theorem 3. *If a top-level variable's value, or a value reachable through arbitrarily many dereferences starting from this value, may be updated during the lifetime of some packet processing procedure, then our analysis marks this top-level variable as updateable.*

Proof. According to the language semantics, scalar and compound values can only be updated via assignment statements. According to Theorem 1, we identify all packet processing procedures. Therefore, identifying all assignment statements in these procedures is sufficient to identify all possible value updates that may occur during the lifetime of some packet processing procedure.

The language semantics also state that the variable on the left-hand-side of an assignment is the variable whose value is updated. Thus, when a top-level variable appears on the left-hand-side of an assignment, we know its value, or a reachable value, is updated. Furthermore, flow-insensitive context-insensitive pointer alias is provably guaranteed to identify all possible points-to relationships [9]. Therefore, any assignment to a variable that may point to a value also pointed to (indirectly) by a top-level variable is identified, and the top-level variable marked updateable. □

Identifying Flowspaces

Theorem 4. *If a middlebox uses standard patterns for fetching values from data structures, and the flowspace for a top-level variable's value (or a value reachable through arbitrarily many dereferences starting from this value) is not constrained by a particular header field, then our analysis does not include this header field in the flowspace fields for this top-level variable.*

Proof. A header field can only be part of a value's flowspace definition if there is a data or control dependency between that header field in the current packet and the fetching of an entry from a data structure. It follows from the proven soundness and precision of flow-sensitive context-insensitive pointer analysis [11] that the SDG will not include false data or control dependency edges. It also follows from the proven soundness of program slicing [18] that only data and control dependencies between source variables (i.e., the packet variable) and target variables (i.e., the index variable, increment variable, or variable in a conditional inside a loop) will be included in the chop. □

Identifying Output-Impacting State

Theorem 5. *If a top-level variable's value, or a value reachable through arbitrarily many dereferences starting from this value, may affect a call to a packet output function or the output produced by the function, then our analysis marks this top-level variable as impacting packet output.*

Proof. Follows from SDG construction soundness [15, 18]. If/when a packet output function is called is determined by a sequence of conditional statements. The path taken at each conditional depends on the values used in the condition. Control and data dependency edges in a system dependence graph capture these features. Since SDG construction is sound [15, 18], we will identify all such dependencies, and thus all values that may affect a call to a packet output function.

Only parameter values, or values reachable through arbitrarily many dereferences starting from these values, can affect the output produced by a packet output function. Thus, knowing what values a parameter value depends on is sufficient to know what values affect the output produced by an output function. Again, since SDG construction is sound, we will identify all such dependencies. □

Embark: Securely Outsourcing Middleboxes to the Cloud

Chang Lan Justine Sherry Raluca Ada Popa Sylvia Ratnasamy Zhi Liu[*]
UC Berkeley [*]*Tsinghua University*

Abstract

It is increasingly common for enterprises and other organizations to outsource network processing to the cloud. For example, enterprises may outsource firewalling, caching, and deep packet inspection, just as they outsource compute and storage. However, this poses a threat to enterprise confidentiality because the cloud provider gains access to the organization's traffic.

We design and build Embark, the first system that enables a cloud provider to support middlebox outsourcing while maintaining the client's confidentiality. Embark encrypts the traffic that reaches the cloud and enables the cloud to process the encrypted traffic without decrypting it. Embark supports a wide-range of middleboxes such as firewalls, NATs, web proxies, load balancers, and data exfiltration systems. Our evaluation shows that Embark supports these applications with competitive performance.

1 Introduction

Middleboxes such as firewalls, NATs, and proxies, have grown to be a vital part of modern networks, but are also widely recognized as bringing significant problems including high cost, inflexibility, and complex management. These problems have led both research and industry to explore an alternate approach: moving middlebox functionality out of dedicated boxes and into software applications that run multiplexed on commodity server hardware [53, 52, 54, 29, 37, 28, 27, 14, 8]. This approach – termed Network Function Virtualization (NFV) in industry – promises many advantages including the cost benefits of commodity infrastructure and outsourced management, the efficiency of statistical multiplexing, and the flexibility of software solutions. In a short time, NFV has gained a significant momentum with over 270 industry participants [27] and a number of emerging product offerings [1, 7, 6].

Leveraging the above trend, several efforts are exploring a new model for middlebox deployment in which a third-party offers middlebox processing as a *service*. Such a service may be hosted in a public cloud [54, 13, 17] or in private clouds embedded within an ISP infrastructure [14, 11]. This service model allows customers such as enterprises to "outsource" middleboxes from their networks entirely, and hence promises many of the known benefits of cloud computing such as decreased costs and ease of management.

However, outsourcing middleboxes brings a new challenge: the confidentiality of the traffic. Today, in order to process an organization's traffic, the cloud sees the traffic *unencrypted*. This means that the cloud now has access to potentially sensitive packet payloads and headers. This is worrisome considering the number of documented data breaches by cloud employees or hackers [23, 60]. Hence, an important question is: can we enable a third party to process traffic for an enterprise, *without seeing the enterprise's traffic*?

To address this question, we designed and implemented Embark[1], the first system to allow an enterprise to outsource a wide range of enterprise middleboxes to a cloud provider, while keeping its network traffic confidential. Middleboxes in Embark operate directly over *encrypted* traffic without decrypting it.

In previous work, we designed a system called BlindBox to operate on encrypted traffic for a *specific* class of middleboxes: Deep Packet Inspection (DPI) [55] – middleboxes that examine only the payload of packets. However, BlindBox is far from sufficient for this setting because (1) it has a restricted functionality that supports too few of the middleboxes typically outsourced, and (2) it has prohibitive performance overheads in some cases. We elaborate on these points in §2.4.

Embark supports a wide range of middleboxes with practical performance. Table 1 shows the relevant middleboxes and the functionality Embark provides. Embark achieves this functionality through a combination of systems and cryptographic innovations, as follows.

From a cryptographic perspective, Embark provides a new and fast encryption scheme called PrefixMatch to enable the provider to perform prefix matching (*e.g.*, if an IP address is in the subdomain 56.24.67.0/16) or port range detection (*e.g.*, if a port is in the range 1000-2000). PrefixMatch allows matching an encrypted packet field against an encrypted prefix or range using the same operators as for unencrypted data: $\geq$ and prefix equality. At the same time, the comparison operators do not work when used between encrypted packet fields. Prior to PrefixMatch, there was no mechanism that provided the functionality, performance, and security needed in our setting. The closest practical encryption schemes are Order-Preserving Encryption (OPE) [21, 48]. However, we show that these schemes are four orders of magnitude slower than

[1] This name comes from "mb" plus "ark", a shortcut for middlebox and a synonym for protection, respectively.

	Middlebox	Functionality	Support	Scheme
L3/L4 Header	IP Firewall [66]	$(SIP, DIP, SP, DP, P) \in (SIP[], DIP[], SP[], DP[], P)$ $\Leftrightarrow \mathsf{Enc}(SIP, DIP, SP, DP, P) \in \mathsf{Enc}(SIP[], DIP[], SP[], DP[], P)$	Yes	PrefixMatch
	NAT (NAPT) [57]	$(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2)$ $\Rightarrow \mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)$ $\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2) \Rightarrow (SIP_1, SP_1) = (SIP_2, SP_2)$	Yes	PrefixMatch
	L3 LB (ECMP) [58]	$(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2)$ $\Leftrightarrow \mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) = \mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2)$	Yes	PrefixMatch
	L4 LB [4]	$(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2)$ $\Leftrightarrow \mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) = \mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2)$	Yes	PrefixMatch
HTTP	HTTP Proxy / Cache [25, 4, 10]	$\mathsf{Match}(\text{Request-URI, HTTP Header})$ $= \mathsf{Match}'(\mathsf{Enc}(\text{Request-URI}), \mathsf{Enc}(\text{HTTP Header}))$	Yes	KeywordMatch
Deep Packet Inspection (DPI)	Parental Filter [10]	$\mathsf{Match}(\text{Request-URI, HTTP Header})$ $= \mathsf{Match}'(\mathsf{Enc}(\text{Request-URI}), \mathsf{Enc}(\text{HTTP Header}))$	Yes	KeywordMatch
	Data Exfiltration / Watermark Detection [56]	$\mathsf{Match}(\text{Watermark, Stream})$ $= \mathsf{Match}'(\mathsf{Enc}(\text{Watermark}), \mathsf{Enc}(\text{Stream}))$	Yes	KeywordMatch
	Intrusion Detection [59, 47]	$\mathsf{Match}(\text{Keyword, Stream}) =$ $\mathsf{Match}'(\mathsf{Enc}(\text{Keyword}), \mathsf{Enc}(\text{Stream}))$	Yes	KeywordMatch
		$\mathsf{RegExpMatch}(\text{RegExp, Stream})$ $= \mathsf{RegExpMatch}'(\mathsf{Enc}(\text{RegExp}), \mathsf{Enc}(\text{Stream}))$	Partially	KeywordMatch
		Run scripts, cross-flow analysis, or other advanced (e.g. statistical) tools	No	-

Table 1: Middleboxes supported by Embark. The second column indicates an encryption functionality that is sufficient to support the core functionality of the middlebox. Appendix §A demonstrates this sufficiency. "Support" indicates whether Embark supports this functionality and "Scheme" is the encryption scheme Embark uses to support it. **Legend:** Enc denotes a generic encryption protocol, SIP = source IP address, DIP = destination IP, SP = source port, DP = destination port, P = protocol, $E[]$ = a range of E, $\Leftrightarrow$ denotes "if and only if", $\mathsf{Match}(x, s)$ indicates if x is a substring of s, and Match' is the encrypted equivalent of Match. Thus, (SIP, DIP, SP, DP, P) denotes the tuple describing a connection.

PrefixMatch making them infeasible for our network setting. At the same time, PrefixMatch provides stronger security guarantees than these schemes: PrefixMatch does not reveal the order of encrypted packet fields, while OPE reveals the total ordering among all fields. We designed PrefixMatch specifically for Embark's networking setting, which enabled such improvements over OPE.

From a systems design perspective, one of the key insights behind Embark is to keep packet formats and header classification algorithms unchanged. An encrypted IP packet is structured just as a normal IP packet, with each field (e.g., source address) containing an encrypted value of that field. This strategy ensures that encrypted packets never appear invalid, e.g., to existing network interfaces, forwarding algorithms, and error checking. Moreover, due to PrefixMatch's functionality, header-based middleboxes can run existing highly-efficient packet classification algorithms [34] without modification, which are among the more expensive tasks in software middleboxes [52]. Furthermore, even software-based NFV deployments use some hardware forwarding components, *e.g.* NIC multiqueue flow hashing [5], 'whitebox' switches [12], and error detection in NICs and switches [5, 2]; Embark is also compatible with these.

Embark's unifying strategy was to reduce the core functionality of the relevant middleboxes to two basic operations over different fields of a packet: prefix and keyword matching, as listed in Table 1. This results in an encrypted packet that *simultaneously* supports these middleboxes.

We implemented and evaluated Embark on EC2. Embark supports the core functionality of a wide-range of middleboxes as listed in Table 1, and elaborated in Appendix A. In our evaluation, we showed that Embark supports a real example for each middlebox category in Table 1. Further, Embark imposes negligible throughput overheads at the service provider: for example, a single-core firewall operating over encrypted data achieves 9.8Gbps, equal to the same firewall over unencrypted data. Our enterprise gateway can tunnel traffic at 9.6 Gbps on a single core; a single server can easily support 10Gbps for a small-medium enterprise.

2 Overview

In this section, we present an overview of Embark.

2.1 System Architecture

Embark uses the same architecture as APLOMB [54], a system which redirects an enterprise's traffic to the cloud for middlebox processing. Embark augments this architecture with confidentiality protection.

In the APLOMB setup, there are two parties: the enterprise(s) and the service provider or cloud (SP). The enterprise runs a gateway (GW) which sends traffic to middleboxes (MB) running in the cloud; in practice, this cloud may be either a public cloud service (such as EC2), or an ISP-supported service running at a Central Office (CO).

We illustrate the two redirection setups from APLOMB in Fig. 1. The first setup, in Fig. 1(a), occurs when the enterprise communicates with an external site: traffic goes to the cloud and back before it is sent out to the Internet. It is worth mentioning that APLOMB allows an optimization that saves on bandwidth and latency relative to Fig. 1(a): the traffic from SP can go directly to the external site and does not have to go back through the gateway. Embark does not allow this optimization fundamentally: the traffic from SP is encrypted and cannot be understood by an external site. Nonetheless, as we demonstrate in §6, for ISP-based deployments this overhead is negligible. For traffic within the same enterprise, where the key is known by two gateways owned by the same company, we can support the optimization as shown in Fig. 1(b).

We do not delve further into the details and motivation of APLOMB's setup, but instead refer the reader to [54].

2.2 Threat Model

Clients adopt cloud services for decreased cost and ease of management. Providers are known and trusted to provide good service. However, while clients trust cloud providers to perform their services correctly, there is an increasing concern that cloud providers may access or leak confidential data in the process of providing service. Reports in the popular press describe companies selling customer data to marketers [20], disgruntled employees snooping or exporting data [16], and hackers gaining access to data on clouds [60, 23]. This type of threat is referred to as an 'honest but curious' or 'passive' [33] attacker: a party who is trusted to handle the data and deliver service correctly, but who looks at the data, and steals or exports it. Embark aims to stop these attackers. Such an attacker differs from the 'active' attacker, who manipulates data or deviates from the protocol it is supposed to run [33]. We consider that such a passive attacker has gained access to *all the data at SP*. This includes any traffic and communication SP receives from the gateway, any logged information, cloud state, and so on.

We assume that the gateways are managed by the enterprise and hence trusted; they do not leak information.

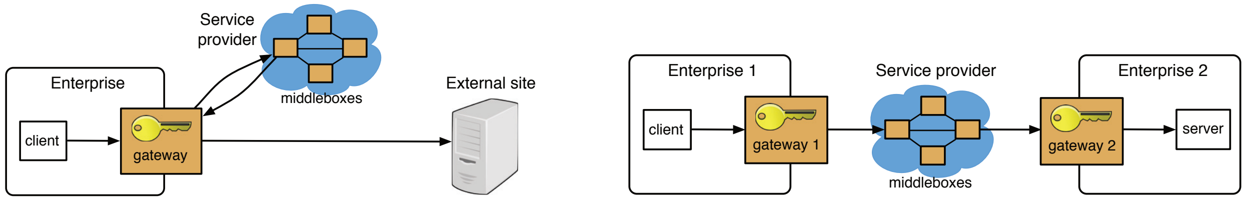
Some middleboxes (such as intrusion or exfiltration detection) have a threat model of their own about the two endpoints communicating. For example, intrusion detection assumes that one of the endpoints could misbehave, but at most one of them misbehaves [47]. We preserve these threat models unchanged. These applications rely on the middlebox to detect attacks in these threat models. Since we assume the middlebox executes its functions correctly and Embark preserves the functionality of these middleboxes, these threat models are irrelevant to the protocols in Embark, and we will not discuss them again.

2.3 Encryption Overview

To protect privacy, Embark *encrypts the traffic* passing through the service provider (SP). Embark encrypts both the header and the payload of each packet, so that SP does not see this information. We encrypt headers because they contain information about the endpoints.

Embark also provides the cloud provider with a set of *encrypted rules*. Typically, header policies like firewall rules are generated by a local network administrator. Hence, the gateway knows these rules, and these rules may or may not be hidden from the cloud. DPI and filtering policies, on the other hand, may be private to the enterprise (as in exfiltration policies), known by both parties (as in public blacklists), or known only by the cloud provider (as in proprietary malware signatures). We discuss how rules are encrypted, generated and distributed given these different trust settings in §4.2.

As in Fig. 1, the gateway has a secret key k; in the setup with two gateways, they share the same secret key. At setup time, the gateway generates the set of encrypted rules using k and provides them to SP. Afterwards, the gateway encrypts all traffic going to the service provider using Embark's encryption schemes. The middleboxes at SP process encrypted traffic, comparing the traffic against the encrypted rules. After the processing, the middleboxes will produce encrypted traffic which SP sends back to the



(a) Enterprise to external site communication

(b) Enterprise to enterprise communication

Figure 1: System architecture. APLOMB and NFV system setup with Embark encryption at the gateway. The arrows indicate traffic from the client to the server; the response traffic follows the reverse direction.

gateway. The gateway decrypts the traffic using the key k.

Throughout this process, middleboxes at SP handle only encrypted traffic and never access the decryption key. On top of Embark's encryption, the gateway can use a secure tunneling protocol, such as SSL or IPSec to secure the communication to SP.

Packet encryption. A key idea is to encrypt packets *field-by-field*. For example, an encrypted packet will contain a source address that is an encryption of the original packet's source address. We ensure that the encryption has the same size as the original data, and place any additional encrypted information or metadata in the options field of a packet. Embark uses three encryption schemes to protect the privacy of each field while allowing comparison against encrypted rules at the cloud:

- Traditional AES: provides strong security and no computational capabilities.
- KeywordMatch: allows the provider to detect if an encrypted value in the packet is equal to an encrypted rule; does not allow two encrypted values to be compared to each other.
- PrefixMatch: allows the provider to detect whether or not an encrypted value lies in a range of rule values – e.g. addresses in 128.0.0.0/24 or ports between 80-96.

We discuss these cryptographic algorithms in §3.

For example, we encrypt IP addresses using PrefixMatch. This allows, e.g., a firewall to check whether the packet's source IP belongs to a prefix known to be controlled by a botnet – but without learning what the actual source IP address is. We choose which encryption scheme is appropriate for each field based on a classification of middlebox capabilities as in Table 1. In the same table, we classify middleboxes as operating only over L3/L4 headers, operating only over L3/L4 headers and HTTP headers, or operating over the entire packet including arbitrary fields in the connection bytestream (DPI). We revisit each category in detail in §5.

All encrypted packets are IPv6 because PrefixMatch requires more than 32 bits to encode an encrypted IP address and because we expect more and more service providers to be moving to IPv6 by default in the future. This is a trivial requirement because it is easy to convert from IPv4 to IPv6 (and back) [42] at the gateway. Clients may continue using IPv4 and the tunnel connecting the gateway to the provider may be either v4 or v6.

Example. Fig. 2 shows the end-to-end flow of a packet through three example middleboxes in the cloud, each middlebox operating over an encrypted field. Suppose the initial packet was IPv4. First, the gateway converts the packet from IPv4 to IPv6 and encrypts it. The options field now contains some auxiliary information which will help the gateway decrypt the packet later. The packet passes through the firewall which tries to match the encrypted information from the header against its encrypted rule, and decides to allow the packet. Next, the exfiltration device checks for any suspicious (encrypted) strings in data encrypted for DPI and not finding any, it allows the packet to continue to the NAT. The NAT maps the source IP address to a different IP address. Back at the enterprise, the gateway decrypts the packet, except for the source IP written by the NAT. It converts the packet back to IPv4.

2.4 Architectural Implications and Comparison to BlindBox

When compared to BlindBox, Embark provides broader functionality and better performance. Regarding functionality, BlindBox [55] enables equality-based operations on encrypted payloads of packets, which supports certain DPI devices. However, this excludes middleboxes such as firewalls, proxies, load balancers, NAT, and those DPI devices that also examine packet headers, because these need an encryption that is compatible with packet headers and/or need to perform range queries or prefix matching.

The performance improvement comes from the different architectural setting of Embark, which provides a set of interesting opportunities. In BlindBox, two arbitrary user endpoints communicate over a modified version of HTTPS. BlindBox requires 97 seconds to perform the initial handshake, which must be performed for every new connection. However, in the Embark context, this exchange can be performed just once at the gateway because the connection between the gateway and the cloud provider is long-lived. Consequently, there is no per-user-connection overhead.

The second benefit is increased deployability. In Embark, the gateway encrypts traffic whereas in BlindBox

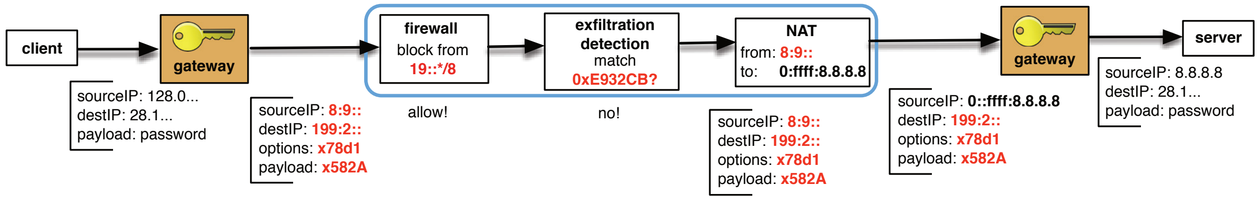


Figure 2: Example of packet flow through a few middleboxes. Red in bold indicates encrypted data.

the end hosts do. Hence, deployability improves because the end hosts do not need to be modified.

Finally, security improves in the following way. BlindBox has two security models: a stronger one to detect rules that are 'exact match' substrings, and a weaker one to detect rules that are regular expressions. The more rules there are, the higher the per-connection setup cost is. Since there is no per-connection overhead in Embark, we can afford having more rules. Hence, we convert many regular expressions to a set of exact-match strings. For example /hello[1-3]/ is equivalent to exact matches on "hello1", "hello2", "hello3". Nonetheless, many regular expressions remain too complex to do so – if the set of potential exact matches is too large, we leave it as a regular expression. As we show in §6, this approach halves the number of rules that require using the weaker security model, enabling more rules in the stronger security model.

In the rest of the paper, we do not revisit these architectural benefits, but focus on Embark's new capabilities that allow us to outsource a *complete* set of middleboxes.

2.5 Security guarantees

We formalize and prove the overall guarantees of Embark in our extended paper. In this version, we provide only a high-level description. Embark hides the values of header and payload data, but reveals some information desired for middlebox processing. The information revealed is the union of the information revealed by PrefixMatch and KeywordMatch, as detailed in §3. Embark reveals more than is strictly necessary for the functionality, but it comes close to this necessary functionality. For example, a firewall learns if an encrypted IP address matches an encrypted prefix, without learning the value of the IP address or the prefix. A DPI middlebox learns whether a certain byte offset matches any string in a DPI ruleset.

3 Cryptographic Building Blocks

In this section, we present the building blocks Embark relies on. Symmetric-key encryption (based on AES) is well known, and we do not discuss it here. Instead, we briefly discuss KeywordMatch (introduced by [55], to which we refer the reader for details) and more extensively discuss PrefixMatch, a new cryptographic scheme we designed for this setting. When describing these schemes, we refer to the encryptor as the gateway whose secret key is k and to the entity computing on the encrypted data as the service provider (SP).

3.1 KeywordMatch

KeywordMatch is an encryption scheme using which SP can check if an encrypted rule (the "keyword") matches by equality an encrypted string. For example, given an encryption of the rule "malicious", and a list of encrypted strings [Enc("alice"), Enc("malicious"), Enc("alice")], SP can detect that the rule matches the second string, but it does not learn anything about the first and third strings, not even that they are equal to each other. KeywordMatch provides typical searchable security guarantees, which are well studied: at a high level, given a list of encrypted strings, and an encrypted keyword, SP does not learn anything about the encrypted strings, other than which strings match the keyword. The encryption of the strings is *randomized*, so it does not leak whether two encrypted strings are equal to each other, unless, of course, they both match the encrypted keyword. We use the scheme from [55] and hence do not elaborate on it.

3.2 PrefixMatch

Many middleboxes perform detection over *prefixes* or *ranges* of IP addresses or port numbers (i.e. packet classification). To illustrate PrefixMatch, we use IP addresses (IPv6), but the scheme works with ports and other value domains too. For example, a network administrator might wish to block access to all servers hosted by MIT, in which case the administrator would block access to the prefix 0::ffff:18.0.0.0/104, *i.e.*, 0::ffff:18.0.0.0/104–0::ffff:18.255.255.255/104. PrefixMatch enables a middlebox to tell whether an encrypted IP address v lies in an encrypted range $[s_1, e_1]$, where s_1 = 0::ffff:18.0.0.0/104 and e_1 = 0::ffff:18.255.255.255/104. At the same time, the middlebox does not learn the values of v, s_1, or e_1.

One might ask whether PrefixMatch is necessary, or one can instead employ KeywordMatch using the same expansion technique we used for some (but not all) regexps in §2.4. To detect whether an IP address is in a range, one could enumerate all IP addresses in that range and perform an equality check. However, the overhead of using this technique for common network ranges such as firewall rules is prohibitive. For our own department network, doing so would convert our IPv6 and IPv4 firewall rule set of only 97 range-based rules to

2^{238} exact-match rules; looking only at IPv4 rules would still lead to 38M exact-match rules. Hence, for efficiency, we need a new scheme for matching ranges.

Requirements. Supporting the middleboxes from Table 1 and meeting our system security and performance requirements entail the following requirements in designing PrefixMatch. First, PrefixMatch must allow for direct order comparison (i.e., using $\leq / \geq$) between an encrypted value $\mathsf{Enc}(v)$ and the encrypted endpoints $\overline{s_1}$ and $\overline{e_1}$ of a range, $[s_1, e_1]$. This allows existing packet classification algorithms, such as tries, area-based quadtrees, FIS-trees, or hardware-based algorithms [34], to run unchanged.

Second, to support the functionality of NAT as in Table 1, $\mathsf{Enc}(v)$ must be *deterministic within a flow*. Recall that a flow is a 5-tuple of source IP and port, destination IP and port, and protocol. Moreover, the encryption corresponding to two pairs (IP$_1$, port$_1$) and (IP$_2$, port$_2$) must be injective: if the pairs are different, their encryption should be different.

Third, for security, we require that nothing leaks about the value v other than what is needed by the functionality above. Note that Embark's middleboxes do not need to know the order between two encrypted values $\mathsf{Enc}(v_1)$ and $\mathsf{Enc}(v_2)$, but only comparison to endpoints; hence, PrefixMatch does not leak such order information. PrefixMatch also provides protection for the endpoints of ranges: SP should not learn their values, and SP should not learn the ordering of the intervals. Further, note that the NAT does not require that $\mathsf{Enc}(v)$ be deterministic across flows; hence, PrefixMatch hides whether two IP addresses encrypted as part of different flows are equal or not. In other words, PrefixMatch is randomized across flows.

Finally, both encryption (performed at the gateway) and detection (performed at the middlebox) should be practical for typical middlebox line rates. Our PrefixMatch encrypts in $< 0.5\mu$s per value (as we discuss in §6), and the detection is the same as regular middleboxes based on the $\leq / \geq$ operators.

Functionality. PrefixMatch encrypts a set of ranges or prefixes P_1, ... , P_n into a set of encrypted prefixes. The encryption of a prefix P_i consists of one or more encrypted prefixes: $\overline{P_{i,1}}\ldots,\overline{P_{i,n_i}}$. Additionally, PrefixMatch encrypts a value v into an encrypted value $\mathsf{Enc}(v)$. These encryptions have the property that, for all i,

$$v \in P_i \Leftrightarrow \mathsf{Enc}(v) \in \overline{P_{i,1}} \cup \cdots \cup \overline{P_{i,n_i}}.$$

In other words, the encryption preserves prefix matching.

For example, suppose that encrypting P = 0::ffff:18.0.0.0/104 results in one encrypted prefix $\overline{P}$ = 1234::/16, encrypting v_1 = 0::ffff:18.0.0.2 results in $\overline{v_1}$ = 1234:db80:85a3:0:0:8a2e:37a0:7334, and encrypting v_2 = 0::ffff:19.0.0.1 results in $\overline{v_2}$ = dc2a:108f:1e16:992e:a53b:43a3:00bb:d2c2. We can see that $\overline{v_1} \in \overline{P}$ and $\overline{v_2} \notin \overline{P}$.

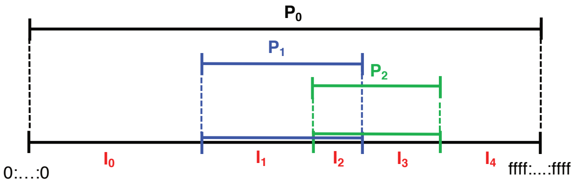


Figure 3: Example of prefix encryption with PrefixMatch.

3.2.1 Scheme

PrefixMatch consists of two algorithms: EncryptPrefixes to encrypt prefixes/ranges and EncryptValue to encrypt a value v.

Prefixes' Encryption. PrefixMatch takes as input a set of prefixes or ranges $P_1 = [s_1, e_1], \ldots, P_n = [s_n, e_n]$, whose endpoints have size len bits. PrefixMatch encrypts each prefix into a set of encrypted prefixes: these prefixes are prefix_len bits long. As we discuss below, the choice of prefix_len depends on the maximum number of prefixes to be encrypted. For example, prefix_len = 16 suffices for a typical firewall rule set.

Consider all the endpoints s_i and e_i laid out on an axis in increasing order as in Fig. 3. Add on this axis the endpoints of P_0, the smallest and largest possible values, 0 and $2^{\mathsf{len}} - 1$. Consider all the non-overlapping intervals formed by each consecutive pair of such endpoints. Each interval has the property that all points in that interval belong to the same set of prefixes. For example, in Fig. 3, there are two prefixes to encrypt: P_1 and P_2. PrefixMatch computes the intervals I_0, ... , I_4. Two or more prefixes/ranges that overlap in exactly one endpoint define a one-element interval. For example, consider encrypting these two ranges [13::/16, 25::/16] and [25::/16, 27::/16]; they define three intervals: [13::/16, 25::/16-1], [25::/16, 25::/16], [25::/16+1, 27::/16].

Each interval belongs to a set of prefixes. Let $\mathsf{prefixes}(I)$ denote the prefixes of interval I. For example, $\mathsf{prefixes}(I_2) = \{P_0, P_1, P_2\}$.

PrefixMatch now assigns an encrypted prefix to each interval. The encrypted prefix is simply a *random* number of size prefix_len. Each interval gets a different random value, except for intervals that belong to the same prefixes. For example, in Fig. 3, intervals I_0 and I_4 receive the same random number because $\mathsf{prefixes}(I_0) = \mathsf{prefixes}(I_4)$.

When a prefix overlaps partially with another prefix, it will have more than one encrypted prefix because it is broken into intervals. For example, I_1 was assigned a random number of 0x123c and I_2 of 0xabcc. The encryption of P_1 in Fig. 3 will be the pair $(123c::/16, abcc::/16)$.

Since the encryption is a random prefix, the encryption does not reveal the original prefix. Moreover, the fact that intervals pertaining to the same set of prefixes receive the same encrypted number hides where an encrypted value matches, as we discuss below. For example, for an IP address v that does not match either P_1 or P_2, the cloud provider will not learn whether it matches to the left or to the right of $P_1 \cup P_2$ because I_0 and I_4 receive the same encryption. The only information it learns about v is that v does not match either P_1 or P_2.

We now present the EncryptPrefixes procedure, which works the same for prefixes or ranges.

```
**EncryptPrefixes** ($P_1$, ..., $P_n$, prefix_len, len):
  Let $s_i$ and $e_i$ be the endpoints of $P_i$.    // $P_i = [s_i, e_i]$
  Assign $P_0 \leftarrow [0, 2^{\mathsf{len}} - 1]$
  Sort all endpoints in $\cup_i P_i$ in increasing order
  Construct non-overlapping intervals $I_0$, ... , $I_m$ from the endpoints as explained above. For each interval $I_i$, compute prefixes($I_i$), the list of prefixes $P_{i_1}, \ldots, P_{i_m}$ that contain $I_i$.
  Let $\overline{I_0}, \ldots, \overline{I_m}$ each be a distinct random value of size prefix_len.
  For all $i, j$ with $i < j$ if prefixes($I_i$) = prefixes($I_j$), set $\overline{I_j} \leftarrow \overline{I_i}$
  The encryption of $P_i$ is $\overline{P_i} = \{\overline{I_j}/\mathsf{prefix\_len}$, for all $j$ s.t. $P_i \in \mathsf{prefixes}(I_j)\}$. The encrypted prefixes are output sorted by value (as a means of randomization).
  Output $\overline{P_1}, \ldots, \overline{P_n}$ and the *interval map* $[I_i \rightarrow \overline{I_i}]$
```

Value Encryption. To encrypt a value v, PrefixMatch locates the one interval I such that $v \in I$. It then looks up $\overline{I}$ in the interval map computed by EncryptPrefixes and sets $\overline{I}$ to be the prefix of the encryption of v. This ensures that the encrypted v, $\overline{v}$, matches $\overline{I}/\mathsf{prefix\_len}$. The suffix of v is chosen at random. The only requirement is that it is deterministic. Hence, the suffix is chosen based on a pseudorandom function [32], $\mathsf{prf}^{\mathsf{suffix\_len}}$, seeded in a given seed seed, where $\mathsf{suffix\_len} = \mathsf{len} - \mathsf{prefix\_len}$. As we discuss below, the seed used by the gateway depends on the 5-tuple of a connection (SIP, SP, DIP, DP, P).

For example, if v is 0::ffff:127.0.0.1, and the assigned prefix for the matched interval is $abcd :: /16$, a possible encryption given the ranges encrypted above is $\mathsf{Enc}(v) = abcd : ef01 : 2345 : 6789 : abcd : ef01 : 2345 : 6789$. Note that the encryption does not retain any information about v other than the interval it matches in because the suffix is chosen (pseudo)randomly. In particular, given two values v_1 and v_2 that match the same interval, the order of their encryptions is arbitrary. Thus, PrefixMatch does not reveal order.

```
**EncryptValue** (seed, $v$, suffix_len, interval map):
  Run binary search on interval map to locate the interval $I$ such that $v \in I$.
  Lookup $\overline{I}$ in the interval map.
  Output
      $$\mathsf{Enc}(v) = \overline{I} \| \mathsf{prf}^{\mathsf{suffix\_len}}_{\mathsf{seed}}(v) \tag{1}$$
```

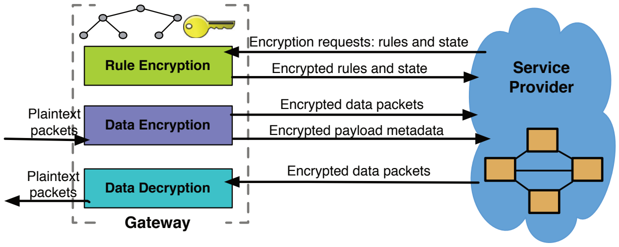


Figure 4: Communication between the cloud and gateway services: rule encryption, data encryption, and data decryption.

Comparing encrypted values against rules. Determining if an encrypted value matches an encrypted prefix is straightforward: the encryption preserves the prefix and a middlebox can use the regular $\leq / \geq$ operators. Hence, a regular packet classification can be run at the firewall with no modification. Comparing different encrypted values that match the same prefix is meaningless, and returns a random value.

3.2.2 Security Guarantees

PrefixMatch hides the prefixes and values encrypted with EncryptPrefixes and EncryptValue. PrefixMatch reveals matching information desired to enable functionality at the cloud provider. Concretely, the cloud provider learns the number of intervals and which prefixes overlap in each interval, but no additional information on the size, order or endpoints of these intervals. Moreover, for every encrypted value v, it learns the indexes of the prefixes that contain v (which is the functionality desired of the scheme), but no other information about v. For any two encrypted values $\mathsf{Enc}(v)$ and $\mathsf{Enc}(v')$, the cloud provider learns if they are equal only if they are encrypted as part of the same flow (which is the functionality desired for the NAT), but it does not learn any other information about their value or order. Hence, PrefixMatch leaks less information than order-preserving encryption, which reveals the order of encrypted prefixes/ranges.

Since EncryptValue is seeded in a per-connection identifier, an attacker cannot correlate values across flows. Essentially, there is a different key per flow. In particular, even though EncryptValue is deterministic within a flow, it is randomized across flows: for example, the encryption of the same IP address in different flows is different because the seed differs per flow.

We formalize and prove the security guarantees of PrefixMatch in our extended paper.

4 Enterprise Gateway

The gateway serves two purposes. First, it redirects traffic to/from the cloud for middlebox processing. Second, it provides the cloud with encryptions of rulesets. Every gateway is configured statically to tunnel traffic to a fixed IP address at a single service provider point of presence. A gateway can be logically thought of as three services: the rule encryption service, the pipeline from the enterprise to the cloud (Data encryption), and the pipeline from the cloud to the enterprise (Data decryption). All three services share access to the PrefixMatch interval map and the private key k. Fig. 4 illustrates these three services and the data they send to and from the cloud provider.

We design the gateway with two goals in mind:

Format-compatibility: in converting plaintext traffic to encrypted traffic, the encrypted data should be structured in such a way that the traffic *appears as normal IPv6 traffic* to middleboxes performing the processing. Format-compatibility allows us to leave fast-path operations unmodified not only in middlebox software, but also in hardware components like NICs and switches; this results in good performance at the cloud.

Scalability and Low Complexity: the gateway should perform only inexpensive per-packet operations and should be parallelizable. The gateway should require only a small amount of configuration.

4.1 Data Encryption and Decryption

As shown in Table 1, we categorize middleboxes as Header middleboxes, which operate only on IP and transport headers; DPI middleboxes, which operate on arbitrary fields in a connection bytestream; and HTTP middleboxes, which operate on values in HTTP headers (these are a subclass of DPI middleboxes). We discuss how each category of data is encrypted/decrypted in order to meet middlebox requirements as follows.

4.1.1 IP and Transport Headers

IP and Transport Headers are encrypted field by field (*e.g.*, a source address in an input packet results in an encrypted source address field in the output packet) with PrefixMatch. We use PrefixMatch for these fields because many middleboxes perform analysis over prefixes and ranges of values – e.g., a firewall may block all connections from a restricted IP prefix.

To encrypt a value with PrefixMatch's EncryptValue, the gateway seeds the encryption with seed $= \mathsf{prf}_k(SIP,\ SP,\ DIP,\ DP,\ P)$, a function of both the key and connection information using the notation in Table 1. Note that in the system setup with two gateways, the gateways generate the same encryption because they share k.

When encrypting IP addresses, two different IP addresses must not map to the same encryption because this breaks the NAT. To avoid this problem, encrypted IP addresses in Embark must be IPv6 because the probability that two IP addresses get assigned to the same encryption is negligibly low. The reason is that each encrypted prefix contains a large number of possible IP addresses. Suppose we have n distinct firewall rules, m flows and a len-bit space, the probability of a collision is approximately:

$$1-e^{\frac{-m^2(2n+1)}{2^{\mathsf{len}+1}}} \tag{2}$$

Therefore, if $\mathsf{len} = 128$ (which is the case when we use IPv6), the probability is negligible in a realistic setting.

When encrypting ports, it is possible to get collisions since the port field is only 16-bit. However, this will not break the NAT's functionality as long as the IP address does not collide, because NATs (and other middleboxes that require injectivity) consider both IP addresses and ports. For example, if we have two flows with source IP and source ports of (SIP, SP_1) and (SIP, SP_2) with $SP_1 \neq SP_2$, the encryption of SIP will be different in the two flows because the encryption is seeded in the 5-tuple of a connection. As we discuss in Appendix A, the NAT table can be larger for Embark, but the factor is small in practice.

Decryption. PrefixMatch is not reversible. To enable packet decryption, we store the AES-encrypted values for the header fields in the IPv6 options header. When the gateway receives a packet to decrypt, if the values haven't been rewritten by the middlebox (*e.g.*, NAT), it decrypts the values from the options header and restores them.

Format-compatibility. Our modifications to the IP and transport headers place the encrypted prefix match data back into the same fields as the unencrypted data was originally stored; because comparisons between rules and encrypted data rely on $\leq \geq$, just as unencrypted data, this means that operations performing comparisons on IP and transport headers *remain entirely unchanged at the middlebox.* This ensures backwards compatibility with existing software *and hardware* fast-path operations. Because per-packet operations are tightly optimized in production middleboxes, this compatibility ensures good performance at the cloud despite our changes.

An additional challenge for format compatibility is where to place the decryptable AES data; one option would be to define our own packet format, but this could potentially lead to incompatibilities with existing implementations. By placing it in the IPv6 options header, middleboxes can be configured to ignore this data.[2]

4.1.2 Payload Data

The connection bytestream is encrypted with KeywordMatch. Unlike PrefixMatch, the data in all flows is encrypted with the same key k. The reason is that KeywordMatch is randomized and it does not leak equality patterns across flows.

This allows Embark to support DPI middleboxes, such as intrusion detection or exfiltration prevention. These devices must detect whether or not there exists

[2] It is a common misconception that middleboxes are incompatible with IP options. Commercial middleboxes are usually aware of IP options but many administrators *configure* the devices to filter or drop packets with certain kinds of options enabled.

an exact match for an encrypted rule string *anywhere* in the connection bytestream. Because this encrypted payload data is over the *bytestream*, we need to generate encrypted values which span 'between' packet payloads. Searchable Encryption schemes, which we use for encrypted DPI, require that traffic be *tokenized* and that a set of fixed length substrings of traffic be encrypted along a sliding window – e.g., the word malicious might be tokenized into 'malici', 'alicio', 'liciou', 'icious'. If the term 'malicious' is divided across two packets, we may not be able to tokenize it properly unless we reconstruct the TCP bytestream at the gateway. Hence, if DPI is enabled at the cloud, we do exactly this.

After reconstructing and encrypting the TCP bytestream, the gateway transmits the encrypted bytestream over an 'extension', secondary channel that only those middleboxes which perform DPI operations inspect. This channel is not routed to other middleboxes. We implement this channel as a persistent TCP connection between the gateway and middleboxes. The bytestream in transmission is associated with its flow identifier, so that the DPI middleboxes can distinguish between bytestreams in different flows. DPI middleboxes handle both the packets received from the extension channel as well as the primary channel containing the data packets; we elaborate on this mechanism in [55]. Hence, if an intrusion prevention system finds a signature in the extension channel, it can sever or reset connectivity for the primary channel.

Decryption. The payload data is encrypted with AES and placed back into the packet payload – like PrefixMatch, KeywordMatch is not reversible and we require this data for decryption at the gateway. Because the extension channel is not necessary for decryption, it is not transmitted back to the gateway.

Format-compatibility. To middleboxes which only inspect/modify packet headers, encrypting payloads has no impact. By placing the encrypted bytestreams in the extension channel, the extra traffic can be routed past and ignored by middleboxes which do not need this data.

DPI middleboxes which do inspect payloads must be modified to inspect the extension channel alongside the primary channel, as described in [55]; DPI devices are typically implemented in software and these modifications are both straightforward and introduce limited overhead (as we will see in §6).

4.1.3 HTTP Headers

HTTP Headers are a special case of payload data. Middleboxes such as web proxies do not read arbitrary values from packet payloads: the only values they read are the HTTP headers. They can be categorized as DPI middleboxes since they need to examine the TCP bytesteam. However, due to the limitation of full DPI support, we treat these values specially compared to other payload data: we encrypt the entire (untokenized) HTTP URI using a deterministic form of KeywordMatch.

Normal KeywordMatch permits comparison between encrypted values and rules, but not between one value and another value; deterministic KeywordMatch permits two values to be compared as well. Although this is a weaker security guarantee relative to KeywordMatch, it is necessary to support web caching which requires comparisons between different URIs. The cache hence learns the frequency of different URIs, but cannot immediately learn the URI values. This is the only field which we encrypt in the weaker setting. We place this encrypted value in the extension channel; hence, our HTTP encryption has the same format-compatibility properties as other DPI devices.

Like other DPI tasks, this requires parsing the entire TCP bytestream. However, in some circumstances we can extract and store the HTTP headers statelessly; so long as HTTP pipelining is disabled and packet MTUs are standard-sized (>1KB), the required fields will always appear contiguously within a single packet. Given that SPDY uses persistent connections and pipelined requests, this stateless approach does not apply to SPDY.

Decryption. The packet is decrypted as normal using the data stored in the payload; IP options are removed.

4.2 Rule Encryption

Given a ruleset for a middlebox type, the gateway encrypts this ruleset with either KeywordMatch or PrefixMatch, depending on the encryption scheme used by that middlebox as in Table 1. For example, firewall rules are encrypted using PrefixMatch. As a result of encryption, some rulesets expand and we evaluate in §6 by how much. For example, a firewall rule containing an IP prefix that maps to two encrypted prefixes using PrefixMatch becomes two rules, one for each encrypted prefix. The gateway should generate rules appropriately to account for the fact that a single prefix maps to encrypted prefixes. For example, suppose there is a middlebox that counts the number of connections to a prefix P. P maps to 2 encrypted prefixes P_1 and P_2. If the original middlebox rule is '`if` v `in` P `then counter++`', the gateway should generate '`if` v `in` P_1 `or` v `in` P_2 `then counter++`'.

Rules for firewalls and DPI services come from a variety of sources and can have different policies regarding who is or isn't allowed to know the rules. For example, exfiltration detection rules may include keywords for company products or unreleased projects which the client may wish to keep secret from the cloud provider. On the other hand, many DPI rules are proprietary features of DPI vendors, who may allow the provider to learn the rules, but not the client (gateway). Embark supports three different models for KeywordMatch rules which allow clients and providers to share rules as they are comfortable: (a) the client knows the rules, and the provider does not; (b) the provider knows the rule, and the client does not; or (c) both parties know the rules. PrefixMatch rules only supports (a) and (c) – the gateway *must* know the rules to perform encryption properly.

If the client is permitted to know the rules, they encrypt them – either generating a KeywordMatch, AES, or PrefixMatch rule – and send them to the cloud provider. If the cloud provider is permitted to know the rules as well, the client will send these encrypted rules annotated with the plaintext; if the cloud provider is not allowed, the client sends only the encrypted rules in random order.

If the client (gateway) is not permitted to know the rules, we must somehow allow the cloud provider to learn the encryption of each rule with the client's key. This is achieved using a classical combination of Yao's garbled circuits [65] with oblivious transfer [40], as originally applied by BlindBox [55]. As in BlindBox, this exchange only succeeds if the rules are signed by a trusted third party (such as McAffee, Symantec, or EmergingThreats) – the cloud provider should not be able to generate their own rules without such a signature as it would allow the cloud provider to read arbitrary data from the clients' traffic. Unlike BlindBox, this rule exchange occurs exactly once – when the gateway initializes the rule. After this setup, all connections from the enterprise are encrypted with the same key at the gateway.

Rule Updates. Rule updates need to be treated carefully for PrefixMatch. Adding a new prefix/range or removing an existing range can affect the encryption of an existing prefix. The reason is that the new prefix can overlap with an existing one. In the worst case, the encryption of all the rules needs to be updated.

The fact that the encryption of old rules changes poses two challenges. The first challenge is the correctness of middlebox state. Consider a NAT with a translation table containing ports and IP addresses for active connections. The encryption of an IP address with EncryptValue depends on the list of prefixes so an IP address might be encrypted differently after the rule update, becoming inconsistent with the NAT table. Thus, the NAT state must also be updated. The second challenge is a race condition: if the middlebox adopts a new ruleset while packets encrypted under the old ruleset are still flowing, these packets can be misclassified.

To maintain a consistent state, the gateway first runs EncryptPrefixes for the new set of prefixes. Then, the gateway announces to the cloud the pending update, and the middleboxes ship their current state to the gateway. The gateway updates this state by producing new encryptions and sends the new state back to the middleboxes. During all this time, the gateway continued to encrypt traffic based on the old prefixes and the middleboxes processed it based on the old rules. Once all middleboxes have the new state, the gateway sends a signal to the cloud that it is about to 'swap in' the new data. The cloud buffers incoming packets after this signal until all ongoing packets in the pipeline finish processing at the cloud. Then, the cloud signals to all middleboxes to 'swap in' the new rules and state; and finally it starts processing new packets. For per-packet consistency defined in [51], the buffering time is bounded by the packet processing time of the pipeline, which is typically hundreds of milliseconds. However, for per-flow consistency, the buffering time is bounded by the lifetime of a flow. Buffering for such a long time is not feasible. In this case, if the cloud has backup middleboxes, we can use the migration avoidance scheme [43] for maintaining consistency. Note that all changes to middleboxes are in the *control plane*.

5 Middleboxes: Design & Implementation

Embark supports the core functionality of a set of middleboxes as listed in Table 1. Table 1 also lists the functionality supported by Embark. In Appendix A, we review the core functionality of each middlebox and explain why the functionality in Table 1 is sufficient to support these middleboxes. In this section, we focus on implementation aspects of the middleboxes.

5.1 Header Middleboxes

Middleboxes which operate on IP and transport headers only include firewalls, NATs, and L3/L4 load balancers. Firewalls are read-only, but NATs and L4 load balancers may rewrite IP addresses or port values. For header middleboxes, per-packet operations remain unchanged for both read and write operations.

For read operations, the firewall receives a set of encrypted rules from the gateway and compares them directly against the encrypted packets just as normal traffic. Because PrefixMatch supports $\leq$ and $\geq$, the firewall may use any of the standard classification algorithms [34].

For write operations, the middleboxes assign values from an address pool; it receives these encrypted pool values from the gateway during the rule generation phase. These encrypted rules are marked with a special suffix reserved for rewritten values. When the gateway receives a packet with such a rewritten value, it restores the plaintext value from the pool rather than decrypting the value from the options header.

Middleboxes can recompute checksums as usual after they write.

5.2 DPI Middleboxes

We modify middleboxes which perform DPI operations as in BlindBox [55]. The middleboxes search through the encrypted extension channel – not the packet payloads themselves – and block or log the connection if a blacklisted term is observed in the extension. Embark also improves the setup time and security for regular expression rules as discussed in §2.4.

5.3 HTTP Middleboxes

Parental filters and HTTP proxies read the HTTP URI from the extension channel. If the parental filter observes a blacklisted URI, it drops packets that belong to the connection.

The web proxy required the most modification of any middlebox Embark supports; nonetheless, our proxy achieves good performance as we will discuss in §6. The proxy caches HTTP static content (e.g., images) in order to improve client-side performance. When a client opens a new HTTP connection, a typical proxy will capture the client's SYN packet and open a new connection to the client, as if the proxy were the web server. The proxy then opens a second connection in the background to the original web server, as if it were the client. When a client sends a request for new content, if the content is in the proxy's cache, the proxy will serve it from there. Otherwise, the proxy will forward this request to the web server and cache the new content.

The proxy has a map of encrypted file path to encrypted file content. When the proxy accepts a new TCP connection on port 80, the proxy extracts the encrypted URI for that connection from the extension channel and looks it up in the cache. The use of deterministic encryption enables the proxy to use a fast search data structure/index, such as a hash map, unchanged. We have two possible cases: there is a hit or a miss. If there is a cache hit, the proxy sends the encrypted file content from the cache via the existing TCP connection. Even without being able to decrypt IP addresses or ports, the proxy can still accept the connection, as the gateway encrypts/decrypts the header fields transparently. If there is a cache miss, the proxy opens a new connection and forwards the encrypted request to the web server. Recall that the traffic bounces back to gateway before being forwarded to the web server, so that the gateway can decrypt the header fields and payloads. Conversely, the response packets from the web server are encrypted by the gateway and received by the proxy. The proxy then caches and sends the encrypted content back. The content is separated into packets. Packet payloads are encrypted on a per-packet basis. Hence, the gateway can decrypt them correctly.

5.4 Limitations

Embark supports the core functionality of a wide-range of middleboxes, as listed in Table 1, but not all middlebox functionality one could envision outsourcing. We now discuss some examples. First, for intrusion detection, Embark does not support regular expressions that cannot be expanded in a certain number of keyword matches, running arbitrary scripts on the traffic [47], or advanced statistical techniques that correlate different flows studied in the research literature [69].

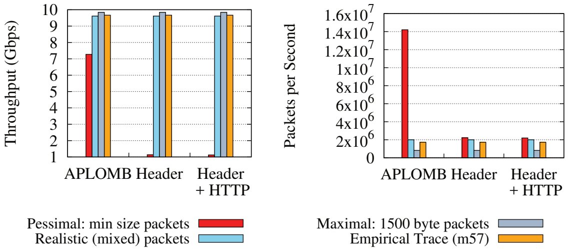


Figure 5: Throughput on a single core at stateless gateway.

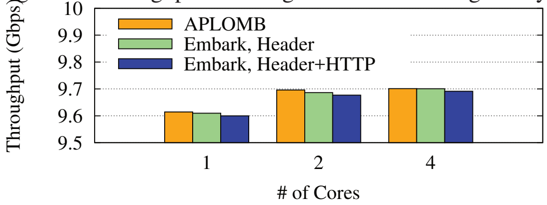


Figure 6: Gateway throughput with increasing parallelism.

Second, Embark does not support application-level middleboxes, such as SMTP firewalls, application-level gateways or transcoders. These middleboxes parse the traffic in an application-specific way – such parsing is not supported by KeywordMatch. Third, Embark does not support port scanning because the encryption of a port depends on the associated IP address. Supporting all these functionalities is part of our future work.

6 Evaluation

We now investigate whether Embark is practical from a performance perspective, looking at the overheads due to encryption and redirection. We built our gateway using BESS (Berkeley Extensible Software Switch, formerly SoftNIC [35]) on an off-the-shelf 16-core server with 2.6GHz Xeon E5-2650 cores and 128GB RAM; the network hardware is a single 10GbE Intel 82599 compatible network card. We deployed our prototype gateway in our research lab and redirected traffic from a 3-server testbed through the gateway; these three client servers had the same hardware specifications as the server we used as our gateway. We deployed our middleboxes on Amazon EC2. For most experiments, we use a synthetic workload generated by the Pktgen [63]; for experiments where an empirical trace is specified we use the m57 patents trace [26] and the ICTF 2010 trace [62], both in IPv4.

Regarding DPI processing which is based on BlindBox, we provide experiment results only for the improvements Embark makes on top of BlindBox, and refer the reader to [55] for detailed DPI performance.

6.1 Enterprise Performance

We first evaluate Embark's overheads at the enterprise.

6.1.1 Gateway

How many servers does a typical enterprise require to outsource traffic to the cloud? Fig. 5 shows the gateway throughput when encrypting traffic to send to the cloud,

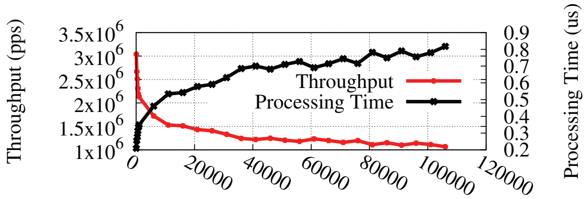


Figure 7: Throughput as # of PrefixMatch rules increases.

first with normal redirection (as used in APLOMB [54]), then with Embark's L3/L4-header encryption, and finally with L3/L4-header encryption as well as stateless HTTP/proxy encryption. For empirical traffic traces with payload encryption (DPI) disabled, Embark averages 9.6Gbps per core; for full-sized packets it achieves over 9.8Gbps. In scalability experiments (Fig. 6) with 4 cores dedicated to processing, our server could forward at up to 9.7Gbps for empirical traffic while encrypting for headers and HTTP traffic. There is little difference between the HTTP overhead and the L3/L4 overhead, as the HTTP encryption only occurs on HTTP requests – a small fraction of packets. With DPI enabled (not shown), throughput dropped to 240Mbps per core, suggesting that an enterprise would need to devote at least 32 cores to the gateway.

How do throughput and latency at the gateway scale with the number of rules for PrefixMatch? In §3.2, we discussed how PrefixMatch stores sorted intervals; every packet encryption requires a binary search of intervals. Hence, as the size of the interval map goes larger, we can expect to require more time to process each packet and throughput to decrease. We measure this effect in Fig. 7. On the y_1 axis, we show the aggregate per packet throughput at the gateway as the number of rules from 0 to 100k. The penalty here is logarithmic, which is the expected performance of the binary search. From 0-10k rules, throughput drops from 3Mpps to 1.5Mpps; after this point the performance penalty of additional rules tapers off. Adding additional 90k rules drops throughput to 1.1Mpps. On the y_2 axis, we measure the processing time per packet, *i.e.*, the amount of time for the gateway to encrypt the packet; the processing time follows the same logarithmic trend.

Is PrefixMatch faster than existing order preserving algorithms? We compare PrefixMatch to BCLO [21] and mOPE [48], two prominent order-preserving encryption schemes. Table 2 shows the results. We can see that PrefixMatch is about four orders of magnitude faster than these schemes.

Operation	**BCLO**	**mOPE**	**PrefixMatch**
Encrypt 10K rules	9333μs	6640μs	0.53μs
Encrypt 100K rules	9333μs	8300μs	0.77μs
Decrypt	169μs	0.128μs	0.128μs

Table 2: PrefixMatch's performance.

What is the memory overhead of PrefixMatch? Storing 10k rules in memory requires 1.6MB, and storing 100k rules in memory requires 28.5MB – using unoptimized C++ objects. This overhead is negligible.

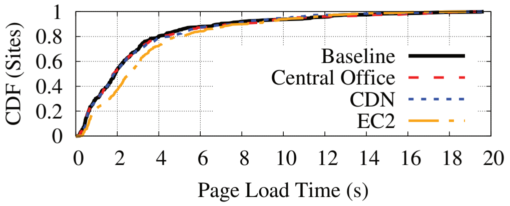


Figure 8: Page load times under different deployments.

6.1.2 Client Performance

We use web performance to understand end-to-end user experience of Embark. Fig. 8 shows a CDF for the Alexa top-500 sites loaded through our testbed. We compare the baseline (direct download) assuming three different service providers: an ISP hosting services in a Central Office (CO), a Content-Distribution Network, and a traditional cloud provider (EC2). The mean RTTs from the gateway are 60μs, 4ms, and 31ms, respectively. We deployed Embark on EC2 and used this deployment for our experiments, but for the CO and CDN we emulated the deployment with inflated latencies and servers in our testbed. We ran a pipeline of NAT, firewall and proxy (with empty cache) in the experiment. Because of the 'bounce' redirection Embark uses, all page load times increase by some fraction; in the median case this increase is less than 50ms for the ISP/Central Office, 100ms for the CDN, and 720ms using EC2; hence, ISP based deployments will escape human perception [39] but a CDN (or a cloud deployment) may introduce human-noticeable overheads.

6.1.3 Bandwidth Overheads

We evaluate two costs: the increase in bandwidth due to our encryption and metadata, and the increase in bandwidth cost due to 'bounce' redirection.

How much does Embark encryption increase the amount of data sent to the cloud? The gateway inflates the size of traffic due to three encryption costs:

- If the enterprise uses IPv4, there is a 20-byte per-packet cost to convert from IPv4 to IPv6. If the enterprise uses IPv6 by default, there is no such cost.
- If HTTP proxying is enabled, there are on average 132 bytes per request in additional encrypted data.
- If HTTP IDS is enabled, there is at worst a 5× overhead on all HTTP payloads [55].

We used the m57 trace to understand how these overheads would play out in aggregate for an enterprise. On the uplink, from the gateway to the middlebox service provider, traffic would increase by 2.5% due to encryption costs for a header-only gateway. Traffic would increase by 4.3× on the uplink for a gateway that supports DPI middleboxes.

How much does bandwidth increase between the gateway and the cloud from using Embark? How much would this bandwidth increase an enterprises' networking costs? Embark sends all network traffic to and from the middlebox service provider for processing, before

Application	Baseline Throughput	Embark Throughput
IP Firewall	9.8Gbps	9.8Gbps
NAT	3.6Gbps	3.5 Gbps
Load Balancer L4	9.8 Gbps	9.8Gbps
Web Proxy	1.1Gbps	1.1Gbps
IDS	85Mbps	166Mbps [55]

Table 3: Middlebox throughput for an empirical workload.

sending that traffic out to the Internet at large.

In ISP contexts, the clients' middlebox service provider and network connectivity provider are one and the same and one might expect costs for relaying the traffic to and from the middleboxes to be rolled into one service 'package;' given the latency benefits of deployment at central offices (as we saw in Fig. 8) we expect that ISP-based deployments are the best option to deploy Embark.

In the cloud service setting the client must pay a third party ISP to transfer the data to and from the cloud, before paying that ISP a third time to actually transfer the data over the network. Using current US bandwidth pricing [24, 38, 61], we can estimate how much outsourcing would increase overall bandwidth costs. Multi-site enterprises typically provision two kinds of networking costs: Internet access, and intra-domain connectivity. Internet access typically has high bandwidth but a lower SLA; traffic may also be sent over shared Ethernet [24, 61]. Intra-domain connectivity usually has a private, virtual Ethernet link between sites of the company with a high SLA and lower bandwidth. Because bounce redirection is over the 'cheaper' link, the overall impact on bandwidth cost with header-only encryption given public sales numbers is between 15-50%; with DPI encryption, this cost increases to between 30-150%.

6.2 Middleboxes

We now evaluate the overheads at each middlebox.

Is throughput reduced at the middleboxes due to Embark?

Table 3 shows the throughput sustained for the apps we implemented. The IP Firewall, NAT, and Load Balancer are all 'header only' middleboxes; the results shown compare packet processing over the same dataplane, once with encrypted IPv6 data and once with unencrypted IPv4 data. The only middlebox for which any overhead is observable is the NAT – and this is a reduction of only 2.7%.

We re-implemented the Web Proxy and IDS to enable the bytestream aware operations they require over our encrypted data. We compare our Web Proxy implementation with Squid [10] to show Embark can achieve competitive performance. The Web Proxy sustains the same throughput with and without encrypted data, but, as we will present later, does have a higher service time per cache hit. The IDS numbers compare Snort (baseline) to the BlindBox implementation; this is not an apples-to-apples comparison as BlindBox performs mostly exact matches where Snort matches regular expressions.

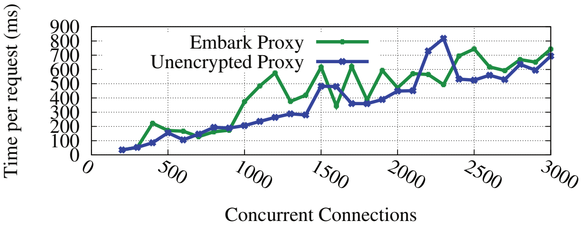


Figure 9: Access time per page against the number of concurrent connections at the proxy.

In what follows, we provide some further middlebox-specific benchmarks for the firewall, proxy, and IDS.

Firewalls: *Does Embark support all rules in a typical firewall configuration? How much does the ruleset "expand" due to encryption?*

We tested our firewall with three rulesets provided to us by a network administrator at our institution and an IP firewall ruleset from Emerging Threats [3]. We were able to encode all rules using range and keyword match encryptions. The size of 3 rulesets did not change after encryption, while the size of the other ruleset from Emerging Threats expanded from 1363 to 1370 – a 0.5% increase. Therefore, we conclude that it has negligible impact on the firewall performance.

Proxy/Caching: The throughput number shown in Table 3 is not the typical metric used to measure proxy performance. A better metric for proxies is how many connections the proxy can handle concurrently, and what time-to-service it offers each client. In Fig. 9, we plot time-to-service against the number of concurrent connections, and see that it is on average higher for Embark than the unencrypted proxy, by tens to hundreds of milliseconds per page. This is not due to computation costs, but instead, due to the fact that the encrypted HTTP header values are transmitted on a different channel than the primary data connection. The Embark proxy needs to synchronize between these two flows; this synchronization cost is what increases the time to service.

Intrusion Detection: Our IDS is based on BlindBox [55]. BlindBox incurs a substantial 'setup cost' every time a client initiates a new connection. With Embark, however, the gateway and the cloud maintain one, long-term persistent connection. Hence, this setup cost is paid once when the gateway is initially configured. Embark also heuristically expands regular expressions in the rulesets into exact match strings. This results in two benefits:

(1) End-to-end performance improvements. Where BlindBox incurs an initial handshake of 97s [55] to open a new connection and generate the encrypted rules, end hosts under Embark never pay this cost. Instead, the gateway pays a one-time setup cost, and end hosts afterwards perform a normal TCP or SSL handshake of only 3-5 RTTs. In our testbed, this amounts to between 30 and 100 ms, depending on the site and protocol – an

improvement of 4 orders of magnitude.

(2) Security improvements. Using IDS rulesets from Snort, we converted regular expressions to exact match strings as discussed in §2.4. In BlindBox, exact match rules can be supported with higher security than regular expressions. With 10G memory, we were able to convert about half of the regular expressions in this ruleset to a finite number of exact match strings; the remainder resulted in too many possible states. We used two rulesets to evaluate this [3, 9]. With the first ruleset BlindBox would resort to a lower security level for 33% of rules, but Embark would only require this for 11.3%. With the second ruleset, BlindBox would use lower security for 58% of rules, but Embark would only do so for 20.2%. At the same time, Embark does not support the lower security level so Embark simply does not support the remaining regexp rules.

It is also worth noting that regular expression expansion in this way makes the one-time setup very slow in one of the three cases: the case when the gateway may not see the rules. The reason is that, in this case, Embark runs the garbled circuit rule-exchange protocol discussed in §4.2, whose slowdown is linear in the number of rules. On one machine, the gateway to server initial setup would take over 3,000 hours to generate the set of encrypted rules due to the large number of keywords. Fortunately, this setup cost is easily parallelizable. Moreover, this setup cost does not occur in the other two rule exchange approaches discussed in §4.2, since they rely only on one AES encryption per keyword rather than a garbled circuit computation which is six orders of magnitude more expensive.

7 Related Work

Middlebox Outsourcing: APLOMB [54] is a practical service for outsourcing enterprise's middleboxes to the cloud, which we discussed in more detail in §2.

Data Confidentiality: Confidentiality of data in the cloud has been widely recognized as an important problem and researchers proposed solutions for software [18], web applications [30, 50], filesystems [19, 36, 31], databases [49, 46], and virtual machines [68]. CryptDB [49] was one of the first practical systems to compute on encrypted data, but its encryption schemes and database system design do not apply to our network setting.

Focusing on traffic processing, the most closely related work to Embark is BlindBox [55], discussed in §2.4. mcTLS [41] proposed a protocol in which client and server can jointly authorize a middlebox to process certain portions of the encrypted traffic. Unlike Embark, the middlebox gains access to *unencrypted data*. A recent paper [67] proposed a system architecture for outsourced middleboxes to specifically perform deep packet inspection over encrypted traffic.

Trace Anonymization and Inference: Some systems which focus on *offline* processing allow some analysis over anonymized data [44, 45]; they are not suitable for online processing as is Embark. Yamada et al [64] show how one can perform some very limited processing on an SSL-encrypted packet by using only the size of data and the timing of packets, however they cannot perform analysis of the contents of connection data.

Encryption Schemes: Embark's PrefixMatch scheme is similar to order preserving encryption schemes [15], but no existing scheme provided both the performance and security properties we required. Order-preserving encryption (OPE) schemes such as [21, 48] are > 10000 times slower than PrefixMatch (§6) and additionally leak the order of the IP addresses encrypted. On the other hand, OPE schemes are more generic and applicable to a wider set of scenarios. PrefixMatch, on the other hand, is designed for our particular scenario.

The encryption scheme of Boneh et al. [22] enables detecting if an encrypted value matches a range and provides a similar security guarantee to PrefixMatch; at the same time, it is orders of magnitude slower than the OPE schemes which are already slower than PrefixMatch.

Acknowledgments

We thank our shepherd, Srinivasan Seshan, and the anonymous reviewers for their thoughtful comments. We're also grateful to Dahlia Malkhi and Ittai Abraham from VMware Research for their valuable feedback on PrefixMatch.

A Sufficient Properties for Middleboxes

In this section, we discuss the core functionality of the IP Firewall, NAT, L3/L4 Load Balancers in Table 1, and why the properties listed in the Column 2 of Table 1 are sufficient for supporting the functionality of those middleboxes. We omit the discussion of other middleboxes in the table since the sufficiency of those properties is obvious. The reason Embark focuses on the core ("textbook") functionality of these middleboxes is that there exist variations and different configurations on these middleboxes and Embark might not support some of them.

A.1 IP Firewall

Firewalls from different vendors may have significantly different configurations and rule organizations, and thus we need to extract a general model of firewalls. We used the model defined in [66], which describes Cisco PIX firewalls and Linux iptables. In this model, the firewall consists of several access control lists (ACLs). Each ACL consists of a list of rules. Rules can be interpreted in the form (*predicate*, *action*), where the *predicate* describes the packets matching this rule and the *action* describes the action performed on the matched packets. The predicate is defined as a combination of ranges of source/destination IP addresses and ports as well as the protocol. The set of possible actions includes "*accept*" and "*deny*".

Let Enc denote a generic encryption protocol, and $(SIP[], DIP[], SP[], DP[], P)$ denote the predicate of a rule. Any packet with a 5-tuple $(SIP, DIP, SP, DP, P) \in (SIP[], DIP[], SP[], DP[], P)$ matches that rule. We encrypt both tuples and rules. The following property of the encryption is sufficient for firewalls.

$$\begin{aligned}&(SIP, DIP, SP, DP, P) \in (SIP[], DIP[], SP[], DP[], P) \Leftrightarrow \\ &\mathsf{Enc}(SIP, DIP, SP, DP, P) \in \\ &\qquad\qquad \mathsf{Enc}(SIP[], DIP[], SP[], DP[], P).\end{aligned} \tag{3}$$

A.2 NAT

A typical NAT translates a pair of source IP and port into a pair of external source IP and port (and back), where the external source IP is the external address of the gateway, and the external source port is arbitrarily chosen. Essentially, a NAT maintains a mapping from a pair of source IP and port to an external port. NATs have the following requirements: 1) same pairs should be mapped to the same external source port; 2) different pairs should not be mapped to the same external source port. In order to satisfy them, the following properties are sufficient:

$$\begin{aligned}&(SIP_1,SP_1) = (SIP_2,SP_2) \\ &\Rightarrow \mathsf{Enc}(SIP_1,SP_1) = \mathsf{Enc}(SIP_2,SP_2),\end{aligned} \tag{4}$$

$$\begin{aligned}&\mathsf{Enc}(SIP_1,SP_1) = \mathsf{Enc}(SIP_2,SP_2) \\ &\Rightarrow (SIP_1,SP_1) = (SIP_2,SP_2).\end{aligned} \tag{5}$$

However, we may relax 1) to: the source IP and port pair that belongs to the same 5-tuple should be mapped to the same external port. After relaxing this requirement, the functionality of NAT is still preserved, but the NAT table may get filled up more quickly since the same pair may be mapped to different ports. However, we argue that this expansion is small in practice because an application on a host rarely connects to different hosts or ports using the same source port. The sufficient properties then become:

$$\begin{aligned}&(SIP_1,DIP_1,SP_1,DP_1,P_1) = (SIP_2,DIP_2,SP_2,DP_2,P_2) \\ &\Rightarrow \mathsf{Enc}(SIP_1,SP_1) = \mathsf{Enc}(SIP_2,SP_2)\end{aligned} \tag{6}$$

and

$$\begin{aligned}&\mathsf{Enc}(SIP_1,SP_1) = \mathsf{Enc}(SIP_2,SP_2) \\ &\Rightarrow (SIP_1,SP_1) = (SIP_2,SP_2).\end{aligned} \tag{7}$$

A.3 L3 Load Balancer

L3 Load Balancer maintains a pool of servers. It chooses a server for an incoming packet based on the L3 connection information. A common implementation of L3 Load Balancing uses the ECMP scheme in the switch. It guarantees that packets of the same flow will be forwarded to the same server by hashing the 5-tuple. Therefore, the sufficient property for L3 Load Balancer is:

$$\begin{aligned}&(SIP_1,DIP_1,SP_1,DP_1,P_1) = (SIP_2,DIP_2,SP_2,DP_2,P_2) \Leftrightarrow \\ &\mathsf{Enc}(SIP_1,DIP_1,SP_1,DP_1,P_1) = \\ &\qquad\qquad \mathsf{Enc}(SIP_2,DIP_2,SP_2,DP_2,P_2).\end{aligned} \tag{8}$$

A.4 L4 Load Balancer

L4 Load Balancer [4], or TCP Load Balancer also maintains a pool of servers. It acts as a TCP endpoint that accepts the client's connection. After accepting a connection from a client, it connects to one of the server and forwards the bytestreams between client and server. The encryption scheme should make sure that two same 5-tuples have the same encryption. In addition, two different 5-tuple should not have the same encryption, otherwise the L4 Load Balancer cannot distinguish those two flows. Thus, the sufficient property of supporting L4 Load Balancer is:

$$\begin{aligned}&(SIP_1,DIP_1,SP_1,DP_1,P_1) = (SIP_2,DIP_2,SP_2,DP_2,P_2) \Leftrightarrow \\ &\mathsf{Enc}(SIP_1,DIP_1,SP_1,DP_1,P_1) = \\ &\qquad\qquad \mathsf{Enc}(SIP_2,DIP_2,SP_2,DP_2,P_2)\end{aligned} \tag{9}$$

B Formal Properties of PrefixMatch

In this section, we show how PrefixMatch supports middleboxes indicated in Table 1. First of all, we formally list the properties that PrefixMatch preserves. As discussed in 3.2, PrefixMatch preserves the functionality of firewalls by guaranteeing Property 3. In addition, PrefixMatch also ensures the following properties:

$$\begin{aligned}&(SIP_1,DIP_1,SP_1,DP_1,P_1) = (SIP_2,DIP_2,SP_2,DP_2,P_2) \Rightarrow\\ &\mathsf{Enc}(SIP_1,DIP_1,SP_1,DP_1,P_1) =\\ &\qquad\qquad \mathsf{Enc}(SIP_2,DIP_2,SP_2,DP_2,P_2)\end{aligned} \tag{10}$$

The following statements hold with *high probability*:

$$\mathsf{Enc}(SIP_1) = \mathsf{Enc}(SIP_2) \Rightarrow SIP_1 = SIP_2 \tag{11}$$

$$\mathsf{Enc}(DIP_1) = \mathsf{Enc}(DIP_2) \Rightarrow DIP_1 = DIP_2 \tag{12}$$

$$\begin{aligned}&\mathsf{Enc}(SIP_1,SP_1) = \mathsf{Enc}(SIP_2,SP_2) \Rightarrow\\ &\qquad (SIP_1,SP_1) = (SIP_2,SP_2)\end{aligned} \tag{13}$$

$$\begin{aligned}&\mathsf{Enc}(DIP_1,DP_1) = \mathsf{Enc}(DIP_2,DP_2) \Rightarrow\\ &\qquad (DIP_1,DP_1) = (DIP_2,DP_2)\end{aligned} \tag{14}$$

$$\mathsf{Enc}(P_1) = \mathsf{Enc}(P_2) \Rightarrow P_1 = P_2 \tag{15}$$

We discuss how those properties imply all the sufficient properties in §A as follows.

NAT We will show that Eq.(10)-Eq.(15) imply Eq.(6)- Eq.(7). Given $(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2)$, by Eq. (10), we have $\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)$. Hence, Eq.(6) holds. Similarly, given $\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)$, by Eq.(13), we have $(SIP_1,SP_1) = (SIP_2,SP_2)$. Hence, Eq.(7) also holds. Note that if we did not relax the property in Eq.(6), we could not obtain such a proof.

L3 Load Balancer By Eq.(10), the left to right direction of Eq.(8) holds. By Eq.(11)-Eq.(15), the right to left direction of Eq.(8) also holds.

L4 Load Balancer By Eq.(10), the left to right direction of Eq.(9) holds. By Eq.(11)-Eq.(15), the right to left direction of Eq.(9) also holds.

References

[1] Brocade Network Function Virtualization. http://www.brocade.com/en/products-services/software-networking/network-functions-virtualization.html.

[2] Cisco IOS IPv6 Commands. http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipv6/command/ipv6-cr-book/ipv6-s2.html.

[3] Emerging Threats.net Open rulesets. http://rules.emergingthreats.net/.

[4] HAProxy. http://www.haproxy.org/.

[5] Intel 82599 10 GbE Controller Datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[6] Network Edge Services Products. https://www.juniper.net/us/en/products-services/network-edge-services/.

[7] Network Function Virtualization for Telecom. http://www.dell.com/learn/us/en/04/tme-telecommunications-solutions-telecom-nfv/.

[8] OPNFV: An Open Platform to Accelerate NFV. https://www.opnfv.org/sites/opnfv/files/pages/files/opnfv_whitepaper_103014.pdf.

[9] Snort v2.9 Community Rules. https://www.snort.org/downloads/community/community-rules.tar.gz.

[10] Squid: Optimising Web Delivery. http://www.squid-cache.org/.

[11] Telefónica NFV Reference Lab. http://www.tid.es/long-term-innovation/network-innovation/telefonica-nfv-reference-lab.

[12] What are White Box Switches? https://www.sdxcentral.com/resources/white-box/what-is-white-box-networking/.

[13] ZScaler. http://www.zscaler.com/.

[14] AT&T Domain 2.0 Vision White Paper. https://www.att.com/Common/about_us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf, Nov. 2013.

[15] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 563–574. ACM, 2004.

[16] Ars Technica. AT&T fined $25 million after call center employees stole customers data. http://arstechnica.com/tech-policy/2015/04/att-fined-25-million-after-call-center-employees-stole-customers-data/.

[17] Aryaka. WAN Optimization. http://www.aryaka.com/.

[18] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 267–283. USENIX Association, 2014.

[19] M. Blaze. A Cryptographic File System for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 9–16. ACM, 1993.

[20] Bloomberg Business. RadioShack Sells Customer Data After Settling With States. http://www.bloomberg.com/news/articles/2015-05-20/radioshack-receives-approval-to-sell-name-to-standard-general.

[21] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-Preserving Symmetric Encryption. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*, EUROCRYPT '09, pages 224–241. Springer-Verlag, 2009.

[22] D. Boneh, A. Sahai, and B. Waters. Fully Collusion Resistant Traitor Tracing with Short Ciphertexts and Private Keys. In *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques*, EUROCRYPT'06, pages 573–592. Springer-Verlag, 2006.

[23] P. R. Clearinghouse. Chronology of data breaches . http://www.privacyrights.org/data-breach.

[24] Comcast. Small Business Internet. http://business.comcast.com/internet/business-internet/plans-pricing.

[25] I. Cooper, I. Melve, and G. Tomlinson. Internet Web Replication and Caching Taxonomy. IETF RFC 3040, Jan. 2001.

[26] Digital Corpora. m57-Patents Scenario. http://digitalcorpora.org/corpora/scenarios/m57-patents-scenario.

[27] European Telecommunications Standards Institute. NFV Whitepaper. https://portal.etsi.org/nfv/nfv_white_paper.pdf.

[28] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using FlowTags. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 533–546. USENIX Association, 2014.

[29] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174. ACM, 2014.

[30] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 47–60. USENIX Association, 2012.

[31] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Tenth Network and Distributed System Security Symposium*, NDSS '03, pages 131–145. Internet Society (ISOC), Feb. 2003.

[32] O. Goldreich. *Foundations of Cryptography: Volume I Basic Tools*. Cambridge University Press, 2001.

[33] M. Goodrich and R. Tamassia. *Introduction to Computer Security*. Pearson, 2010.

[34] P. Gupta and N. McKeown. Algorithms for Packet Classification. *IEEE Network*, 15(2):24–32, Mar. 2001.

[35] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[36] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 29–42. USENIX Association, 2003.

[37] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 459–473. USENIX Association, 2014.

[38] Megapath. Ethernet Data Plus. http://www.megapath.com/promos/ethernet-dataplus/.

[39] R. B. Miller. Response Time in Man-computer Conversational Transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277. ACM, 1968.

[40] M. Naor and B. Pinkas. Efficient Oblivious Transfer Protocols. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 448–457. Society for Industrial and Applied Mathematics, 2001.

[41] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 199–212. ACM, 2015.

[42] E. Nordmark. Stateless IP/ICMP Translation Algorithm (SIIT). IETF RFC 2765, Feb. 2000.

[43] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM.

[44] R. Pang, M. Allman, V. Paxson, and J. Lee. The Devil and Packet Trace Anonymization. *SIGCOMM Computer Communication Review*, 36(1):29–38, Jan. 2006.

[45] R. Pang and V. Paxson. A High-level Programming Environment for Packet Trace Anonymization and Transformation. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 339–351. ACM, 2003.

[46] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind Seer: A Scalable Private DBMS. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 359–374. IEEE Computer Society, 2014.

[47] V. Paxson. Bro: A System for Detecting Network Intruders in Real-time. *Computer Networks*, 31(23-24):2435–2463, Dec. 1999.

[48] R. A. Popa, F. H. Li, and N. Zeldovich. An Ideal-Security Protocol for Order-Preserving Encoding. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 463–477. IEEE Computer Society, 2013.

[49] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100. ACM, 2011.

[50] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building Web Applications on Top of Encrypted Data Using Mylar. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 157–172. USENIX Association, 2014.

[51] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334. ACM, 2012.

[52] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24. USENIX Association, 2012.

[53] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 21:1–21:6. ACM, 2011.

[54] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes

Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 13–24. ACM, 2012.

[55] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 213–226. ACM, 2015.

[56] G. Silowash, T. Lewellen, J. Burns, and D. Costa. Detecting and Preventing Data Exfiltration Through Encrypted Web Sessions via Traffic Inspection. Technical Report CMU/SEI-2013-TN-012, Software Engineering Institute, Carnegie Mellon University, 2013.

[57] P. Srisuresh and K. B. Egevang. Traditional IP Network Address Translator (Traditional NAT). IETF RFC 3022, Jan. 2001.

[58] D. Thaler and C. E. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. IETF RFC 2991, Nov. 2000.

[59] The Snort Project. Snort users manual, 2014. Version 2.9.7.

[60] Verizon. 2015 Data Breach Investigations Report. http://www.verizonenterprise.com/DBIR/2015/.

[61] Verizon. High Speed Internet Packages. http://www.verizon.com/smallbusiness/products/business-internet/broadband-packages/.

[62] G. Vigna. ICTF Data. https://ictf.cs.ucsb.edu/.

[63] K. Wiles. Pktgen. https://pktgen.readthedocs.org/.

[64] A. Yamada, Y. Saitama Miyake, K. Takemori, A. Studer, and A. Perrig. Intrusion Detection for Encrypted Web Accesses. In *21st International Conference on Advanced Information Networking and Applications Workshops*, 2007.

[65] A. C.-C. Yao. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167. IEEE Computer Society, 1986.

[66] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 199–213. IEEE Computer Society, 2006.

[67] X. Yuan, X. Wang, J. Lin, and C. Wang. Privacy-preserving Deep Packet Inspection in Outsourced Middleboxes. In *Proceedings of the 2016 IEEE Conference on Computer Communications*, INFOCOM '16, 2016.

[68] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216. ACM, 2011.

[69] Y. Zhang and V. Paxson. Detecting stepping stones. In *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9*, SSYM'00, pages 13–13. USENIX Association, 2000.

BUZZ: Testing Context-Dependent Policies in Stateful Networks

Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, Vyas Sekar
CMU

Abstract

Checking whether a network correctly implements intended policies is challenging even for basic reachability policies (Can X talk to Y?) in simple stateless networks with L2/L3 devices. In practice, operators implement more complex *context-dependent* policies by composing *stateful* network functions; e.g., if the IDS flags X for sending too many failed connections, then subsequent packets from X must be sent to a deep-packet inspection device. Unfortunately, existing approaches in network verification have fundamental expressiveness and scalability challenges in handling such scenarios. To bridge this gap, we present BUZZ, a practical model-based testing framework. BUZZ's design makes two key contributions: (1) Expressive and scalable models of the data plane, using a novel high-level traffic unit abstraction and by modeling complex network functions as an ensemble of finite-state machines; and (2) A scalable application of symbolic execution to tackle state-space explosion. We show that BUZZ generates test cases for a network with hundreds of network functions within two minutes (five orders of magnitude faster than alternative designs). We also show that BUZZ uncovers a range of both new and known policy violations in SDN/NFV systems.

1 Introduction

The security, performance, and availability of networks depend on the correct implementation of critical policy goals. Network operators realize these goals by configuring and composing network appliances, such as switches/routers, firewalls, and proxies.

Unfortunately, making sure that the network correctly implements a given policy is challenging, error-prone, and entails significant manual effort and operational costs [20,59]. As recent advances in network verification show, checking correctness is challenging even for simple *reachability* policies (Can X talk to Y?) in networks with *stateless* switches and routers [44,52,53,57,75].

In practice, operators' intended policies go well beyond reachability— operators implement a range of rich *context-dependent* policies using *stateful* network functions (NFs)[1] to ensure traffic goes through the intended sequence of NFs; e.g., if an intrusion detection system (IDS) flags host X for generating too many connections (i.e., if traffic context is "alarm"), then reroute subsequent flows to a deep packet inspection (DPI) filter [23]. Such rich policies and stateful data planes are quite common (e.g., the number of stateful NFs in a network may be comparable to the number of routers [70]). Looking forward, software-defined networking (SDN) [60] and network functions virtualization (NFV) [34] are poised to enable even richer in-network traffic processing services [22,26,29,34,42,56].

What is critically lacking today is a principled way to check whether a stateful data plane correctly implements intended context-dependent policies. Existing approaches [44,52,53,57,75] face fundamental *expressiveness* and *scalability* challenges in this regard. First, current abstractions cannot capture stateful behaviors (e.g., how many connections host X has tried to establish) or express context-dependent policies (e.g., on-demand deep inspection). Second, trying to reason about stateful behaviors results in state-space explosion; e.g., a naive application of formal verification tools takes $>$ 20 hours even for a small network with 4-5 nodes (see §8).

We address these challenges and develop a principled testing framework called BUZZ. BUZZ takes in intended policies from the operator, and by exploring a model of the data plane, it finds abstract test traffic (i.e., an input that triggers policy-relevant states of a model of the data plane). It then translates the abstract test traffic into concrete test traffic and injects it into the actual data plane. Finally, it reports whether the observed behavior complies with the policies. As an active testing framework, BUZZ provides concrete assurances about the behavior "on-the-wire" and can help operators localize sources of violations [75] (§3).

In designing BUZZ, we make two key contributions:

- **Expressive-yet-scalable data plane models (§5):** We introduce a novel abstraction for network traffic called a BUZZ Data Unit (BDU). BDUs extend the notion of located packets from prior work [52] in three key ways: (1) it enables composition of diverse NFs spanning multiple protocol layers; (2) it simplifies models of NFs operating above L3 by aggregating a sequence of packets; and (3) it explicitly encodes traffic processing history to expose policy-relevant contexts. Second, we model individual NFs as FSMs that process BDUs and explicitly embed the relevant contexts into BDUs. A network then is simply a composition of individual NF models. To build tractable models, we

[1] An NF may be a switch/router or a middlebox (e.g., firewalls, load balancers, intrusion prevention systems, or proxies). It may be realized by a physical appliance or a virtual machine (VM).

decouple logically independent tasks (e.g., client-side vs. server-side connections) or units of traffic (e.g., distinct TCP connections) within each NF to create an ensemble of FSMs representation rather than a monolithic FSM.

- **Scalable test traffic generation (§6):** To generate abstract test traffic to explore the behaviors of the data plane model, we develop an optimized symbolic execution (SE)-based workflow. To combat the challenge of state space explosion [30,32], we engineer domain-specific optimizations (e.g., reducing the number and scope of symbolic variables). We also develop custom translation mechanisms to convert the output of this step into concrete test traffic.

We have implemented BUZZ as an application over `OpenDaylight` [14]. BUZZ provides both text-based and graphical interfaces for operators to input policies and receive test results through an automated workflow. We have written a library of models for several canonical NFs and implemented our SE optimizations using KLEE [31]. We have also developed simple monitoring and test resolution mechanisms (§7). BUZZ is open-source, and our code, models, and examples can be found at [1].

Our evaluation (§8) on a real testbed shows that BUZZ: (1) effectively helps detect both new and known policy violations within tens of seconds; (2) tests hundreds of policies in networks with hundreds of switches and stateful NFs within two minutes; (3) dramatically improves test scalability, providing nearly five orders of magnitude reduction in time for test traffic generation relative to strawman solutions (e.g., model checking).

2 Motivation

In this section, we use a few illustrative examples to discuss why it is challenging to check the correctness of context-dependent policies in stateful data planes.

Stateful firewalling: Today most firewalls capture TCP semantics. A common usage is reflexive ACLs [5] as shown in Figure 1, where incoming traffic is allowed depending on its *context*. In particular, the context-dependent policy here specifies that only traffic belonging to a TCP connection initiated by a host inside the department (i.e., if traffic context is "solicited") be allowed.

Prior work in network verification models each NF as a "transfer" function $T(hdr, port)$ whose input/output is a located packet (i.e., a *header, port* tuple) (e.g., [52, 53, 62]). Unfortunately, even the simple policy of Figure 1 cannot be captured by this stateless transfer function. In particular, it does not capture the policy-relevant *state* of the firewall (e.g., `SYN_SENT`) for a given connection.

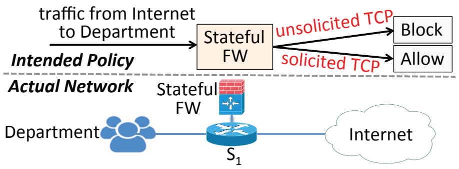


Figure 1: Is firewall allowing solicited and blocking unsolicited traffic?

Context-dependent traffic monitoring: In Figure 2, the operator uses a proxy to improve web performance. She also wants to restrict web access; i.e., H_2 (a host in the department) cannot have access to `XYZ.com`. Here the context-dependent policy specifies that both cache hits/misses for H_2 should be monitored. As noted elsewhere [43], there could be subtle policy violations where cached responses evade the monitor because (1) the proxy hides traffic provenance (i.e., true origin), and (2) the proxy's response (i.e., hit vs. miss) depends on the hidden policy-relevant *state* (i.e., the current cache contents).

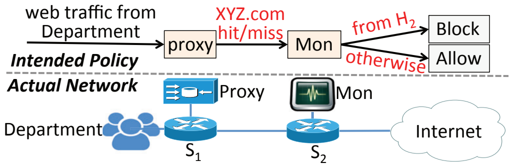


Figure 2: Are both cache hit/miss traffic monitored?

While there are mechanisms to fix this (e.g., [43]), operators need tools to check whether such mechanisms are implemented correctly. Again, a stateless transfer function [52, 53, 57] is insufficient, as it does not capture the state of the proxy.

Multi-stage triggers: Figure 3 uses a light-weight intrusion prevention system (L-IPS) for all traffic, and only subjects suspicious hosts (i.e., flagged by the L-IPS due to generating too many scans) to the expensive heavy-weight IPS (H-IPS) for payload signature matching. Such context-dependent multi-stage detection can minimize latency and reduce H-IPS load [42].

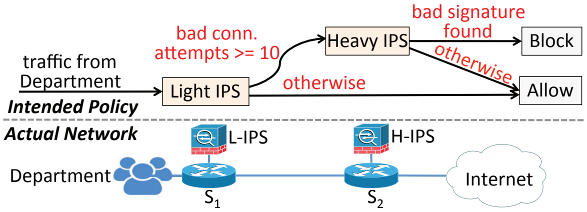


Figure 3: Is suspicious traffic sent to heavy IPS?

Again, we cannot check if such multi-stage policies are enforced correctly using existing mechanisms [44, 52, 53, 75] because they capture neither policy context (e.g., alarm/not alarm) nor data plane state (e.g., the count of bad connection attempts on L-IPS). This example also demonstrates that just capturing packet headers (e.g., [52, 53, 57]) is not sufficient, as the behavior of the H-IPS may depend on packet contents.

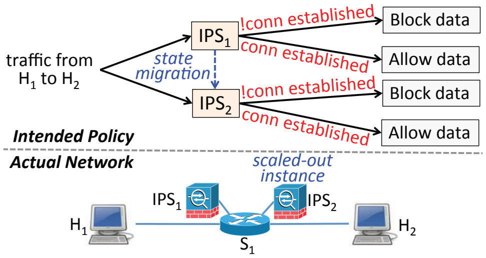


Figure 4: Does the scale-out mechanism honor the stateful semantics of migration?

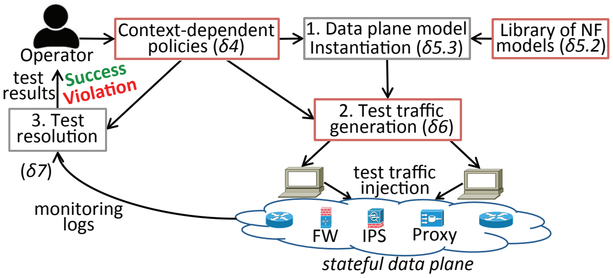


Figure 5: High-level workflow of BUZZ.

Dynamic NF deployments: NFV creates new opportunities for elastic scaling of *NFs* [34]. However, ensuring the correctness of policies in the presence of elastic scaling is not easy. For example, in Figure 4, suppose IPS_1 observes flow f_1 established between the two hosts; later f_1 is migrated to the newly launched IPS_2 for better load balancing [68]. Due to the stateful semantics of the IPS, IPS_2 needs to know that f_1 has already established a TCP connection; otherwise, IPS_2 may incorrectly block this flow. While recent efforts enable state migration [46,68], we need ways to check whether they do so correctly.

Similarly, in dynamic *NF* failure recovery [34], if the main *NF* fails, the backup *NF* needs to be activated with the correct state so that traffic is uninterrupted (e.g., see [69]). Again, we lack the ability to check whether such mechanisms work as intended.

3 Overview

Our goal is to enable network operators to check at human-interactive timescales whether their context-dependent policies are realized in stateful data planes. Next, we present a high-level view of BUZZ to meet this goal and summarize key challenges in realizing it.

To put our work in perspective, we note that there are two complementary approaches: *(1) Static verification* uses network configuration files to check whether the network behavior complies with the intended policies assuming the data plane behaves correctly (e.g., HSA [52], Veriflow [53], NOD [57], Batfish [44]); *(2) Active testing*, on the other hand, checks the behavior of the data plane by injecting test traffic into the network [75]. While both are useful, we adopt an active testing approach for two reasons. First, it provides practical assurances that things are actually working correctly "on-the-wire". Second, network behaviors in certain scenarios such as dynamic *NF* deployment (Figure 4) are hard to capture with a purely static approach.

Due to context-dependent policies and complex stateful behaviors, naive attempts to generate test traffic, either manually or via fuzzing [47,61], are ineffective. For example, in Figure 3, in order to trigger the policy context "L-IPS alarm" and check if traffic will actually go to H-IPS, we need to carefully craft a sequence of packets that drive the count of bad connections on L-IPS to ≥ 10; achieving this via randomly generated packets is unlikely. Our goal is to automate this process.

To bridge the gap between policies and the actual data plane, we adopt model-based testing (MBT) [72], which is useful when the blackbox behavior of a system needs to be actively tested. The high-level idea is to (1) use a *model* (or *specification*) of the system under test and a *search* mechanism to systematically find *test inputs* that trigger certain behaviors of the model, and then (2) compare the behavior of the system under test to the behavior of the model for each input [72].

Figure 5 shows the high-level workflow of BUZZ:

1. *Model Instantiation*: BUZZ instantiates a model of the data plane using the intended policies (the only input by the operator) and a library of *NF* models;
2. *Test Traffic Generation*: BUZZ generates abstract test traffic to trigger policy-relevant behaviors of the data plane model. BUZZ then translates it into concrete test traffic, which is then injected into the actual data plane;
3. *Test Resolution*: BUZZ monitors the actual data plane and compares the observed behavior to the intended policies. The result (i.e., success/violation) is reported to the operator.

There are two challenges in realizing this workflow:

- *Expressive-yet-scalable data plane models:* To see why this is challenging, let us consider some seemingly natural candidates. A natural starting point would be the transfer function abstraction [52, 62]; however, it is not expressive, as it offers no stateful semantics and no binding to the relevant context. On the other hand, using an *NF*'s implementation code as its model is not tractable (e.g., Squid [18] has $\geq$ 200K lines of code) and may suffer from other practical limitations (e.g., code may not be available, or implementation bugs may affect test traffic).
- *Scalable test traffic generation:* Exploring data plane's behaviors is challenging even for simple reachability policies in stateless data planes [75]. Our setting is worse, as reasoning about stateful behaviors requires addressing the challenge of state-space explosion. Off-the-shelf mechanisms (e.g., model checking) struggle beyond a few hundred lines of code (see §6 and §8).

Listing 1: An abstract stateful NF.

```
1  //Input: packet inPkt on port inPort
2  ⟨outPkt, state⟩ ← process(inPkt, state)
3  context ← stateToContextMap(state)
4  outPort ← applyPolicy(outPkt, context)
5  dispatch(outPkt, outPort)
```

We address these two challenges in §5 and §6, respectively. Before doing so, in the next section (§4), we first formalize our problem to shed light on the key requirements of modeling the data plane and generating test traffic.

4 Problem Formulation

In this section, we formalize our model-based testing framework to see what a data plane model should capture and what test traffic needs to do. These inform our approach to modeling (§5) and test traffic generation (§6).

4.1 Intuition behind model and test traffic

What should the data plane model capture: First, we give the intuition behind what an NF model needs to capture. As we saw in §2, data planes are stateful (e.g., the bad connection attempts count in Figure 3). However, being stateful is not sufficient for a data plane model to be expressive. Specifically, to test context-dependent policies, the model needs to explicitly map each state to a context. For example, if we want to trigger an alarm on L-IPS in Figure 3 (e.g., to check if the traffic will actually go to H-IPS), we need to capture the mapping from the bad connection attempts count (e.g., ≥ 10 or < 10) to the context (e.g., alarm or not alarm).

To understand what an NF model should capture, we consider the abstract NF shown in Listing 1 that shows the NF model as running three logical steps: (1) It processes an input packet and updates some relevant state (e.g., an IPS updating `bad_conn_attempts_count`) (Line 2); (2) It extracts the relevant *context* for the processed packet (e.g., alarm on an IPS based on `bad_conn_attempts_count`) (Line 3); (3) It applies the corresponding policy (e.g., drop, forward) via function $applyPolicy(.)$ and then dispatches the packet to the policy-mandated port (Lines 4-5).

What should test traffic do? At a high level, test traffic for a given policy needs to drive the data plane to a state corresponding to the context. In Listing 1, this means we need to find a sequence of packets that drives the NF to a state (Line 2) that maps to the intended context (Line 3). If the NF is policy-compliant, the traffic at this point will be sent to a policy-mandated port (Lines 4-5). For example, to exercise the context of "L-IPS alarm" in Figure 3, test traffic needs to make `bad_conn_attempts_count` to exceed 10; then, we check whether traffic at this point actually goes to H-IPS.

4.2 Formal framework

Having seen the intuition behind state, context, and test traffic, we formalize these to inform our design.

Context-dependent policies: Let $context^{pkt}_{NF_i}$ denote the processing context corresponding to packet pkt at NF_i (Line 3 of Listing 1). Then, the *context sequence* of the packet is the sequence of contexts along the NFs it has traversed; i.e., if pkt has traversed NF_1, ..., NF_i, its context sequence is $ContextSeq^{pkt} = \langle context^{pkt}_{NF_1}, \ldots, context^{pkt}_{NF_i} \rangle$.

Context-dependent policies are expressed as a set of rules of the form:

$$Policy : TrafficSpec \times ContextSeq \mapsto PortSeq$$

Here, *TrafficSpec* is a predicate on the IP 5-tuple (e.g., source IP and transport protocol), *ContextSeq* is a context sequence, and *PortSeq* is a sequence of network ports *Ports* (interfaces).[2] For example, in Figure 3, the policy that mandates "if traffic triggers an alarm on L-IPS, it must be sent to H-IPS" is specified as:

$$\langle \texttt{srcIP=Dept} \rangle, \langle alarm_{L-IPS} \rangle \mapsto$$
$$\langle L{-}IPS \rightarrow S_1, S_1 \rightarrow S_2, S_2 \rightarrow H{-}IPS \rangle$$

(Policies for dynamic NF deployments, such as Figure 4, are defined slightly differently—see §6.4.)

Stateful data planes: Contexts are convenient "shorthands" to define policies. In reality, however, the data plane operates in terms of the related but (possibly) lower-level notion of state.

As we saw in Listing 1, a stateful NF takes an input packet on one of its ports, processes it, goes to a new state, and outputs a packet on one of its ports. A stateful NF can be naturally expressed as a finite-state machine (FSM) of the form $NF_i = (S_i, I_i, Ports_i, T_i)$, where S_i is the set of NF_i states, I_i is the initial state of NF_i, $Ports_i$ is the set of ports of NF_i (where $Ports_i \in Ports$), and $T_i : Pkts \times Ports_i \times S_i \mapsto Pkts \times Ports_i \times S_i$ is the stateful (as opposed to stateless, e.g., [52]) transfer function of NF_i. We model intended packet drops as sending packets to a virtual "drop port" on the NF. To model the entire data plane, the topology function $\tau : Ports \mapsto Ports$ captures the physical interconnection of NFs. Finally, we define the state of the data plane, S_{DP}, as the conjunction of the states of its individual NFs.

There are many levels of abstraction to write such an FSM on, from low-level code variables to high-level logical states (e.g., proxy cache state). Irrespective of this

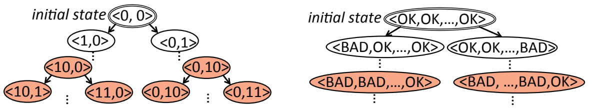
[2] Without loss of generality, we assume policies are in terms of physical NF instances as opposed to logical types of NFs. This is more precise because the semantics of stateful NFs (e.g., NATs) requires that both directions of a flow pass the same NF instance.

granularity, to be expressive for testing the model needs to provide a mapping from the states to the corresponding traffic specification and context:

$$stateToContextMap_i : 2^{S_i} \mapsto TrafficSpec \times C_i$$

where C_i denotes the set of all contexts of NF_i.

To illustrate this, let us revisit Figure 3. Figure 6 shows two possible ways of modeling L-IPS as an FSM. In both Figures 6a and 6b, each of the red states maps to $\langle \texttt{srcIP=Dept} \rangle, \langle alarm_{L-IPS} \rangle$—these mappings make the models expressive. (In §5, we will discuss other requirements of an FSM-based NF model in addition to expressiveness.)



(a) Each state is of the form $\langle badAttmpCnt_{H_1}, badAttmpCnt_{H_2} \rangle$

(b) Each state is of the form $\langle connStatus_{f_1}, \ldots, connStatus_{f_{20}} \rangle$

Figure 6: Two example FSM models of L-IPS of Figure 3 assuming a world with 2 hosts and 20 flows. The states corresponding to alarm (i.e., at least 10 bad connection attempts) are highlighted in red.

Test traffic: Test traffic needs to trigger the policy context by driving the data plane to a state that corresponds the context (e.g., a red state in Figure 6). Thus, $trace = \langle pkt_1, \ldots, pkt_m, \ldots, pkt_r \rangle$ is a test trace for $policy : trafficSpec \times contextSeq \mapsto portSeq$ iff:

1. Each packet $pkt \in trace$ satisfies *trafficSpec*, and
2. S_{DP} does not correspond to *contextSeq* after injection of each of packets $\langle pkt_1, \ldots, pkt_{m-1} \rangle$, and
3. S_{DP} corresponds to *contextSeq* after injection and processing each of packets $\langle pkt_m, \ldots, pkt_r \rangle$.

After $trace$ is injected into the actual data plane, test resolution involves checking whether packets $\langle pkt_m, \ldots, pkt_r \rangle$ actually traverse ports *portSeq*.

Takeaways: This framework suggests two key design implications: (1) While an FSM is a natural starting point to model a stateful NF, an expressive model should bridge the gap between its states and policy-mandated traffic specification and context (§5); and (2) Test traffic should satisfy the traffic specification and drive the data plane to a state that corresponds to the policy context (§6).

5 Data Plane Model Instantiation

In this section, we discuss how to instantiate a model of the data plane. Recall from §3 that this stage takes as input a library of NF models and the policy. The challenge in building such a library is to model each type of NF (e.g., stateful firewall, web proxy) such that these models are (1) *composable*, despite diverse types of NFs operating at different network layers; (2) *expressive*, despite stateful behaviors and hidden context; and (3) *scalable* to explore. After presenting our high-level approach (§5.1), we introduce a new abstract data unit for modeling input-output of NFs and describe how we create scalable NF models via an ensemble-of-FSMs representation (§5.2). Finally, we describe how we construct the network-wide model composing individual models of NFs (§5.3).

5.1 High-level idea

A natural starting point to model an NF that is composable is the *transfer function* from prior work [52, 62]. Each NF is modeled as: $lp \leftarrow T(lp)$. Here, the input/output is a located packet $lp = (pkt, port)$, an IP packet (header) along with its location in the network. However, as we saw in §2, this is not expressive on several fronts w.r.t. state and context. To see how we can make it expressive, let us revisit our abstract NF from Listing 1 and contrast it with the transfer function. This highlights two key missing elements: (1) there is no notion of state, and (2) the located packet has no binding to the relevant context.

Our formalism from §4 suggests two extensions: (1) Instead of the (stateless) transfer function, we need to move to an FSM-like abstraction that captures state and the state-to-context mappings; and (2) We need some way to logically bind a packet to its relevant context. To this end, we extend the located packet abstraction so that it carries the relevant context history as it traverses the data plane model. Then, we can consider an NF as an FSM that processes this extended located packet and explicitly includes the policy-relevant context in the outgoing packet. In a nutshell, this summarizes our basic insight to create an expressive model.

Next, we discuss how we translate this insight into a concrete realization. We also address the scalability requirement of NF models, as a naive FSM model will have too many states to explore.

5.2 Modeling individual NFs

The BUZZ Data Unit (BDU): We start by presenting our approach to modeling the extended located packet idea described above and explain how it enables composability, expressiveness, and scalability. Concretely, a BDU is a *struct* as shown in Listing 2 that extends a located packet [52, 62] in three key ways:

1. *Multi-layer abstraction with IP as the common denominator:* Unlike a located packet, a BDU can explicitly encode higher-layer semantics (e.g., HTTP GET or responses). The key to achieving model composability while enabling higher-layer semantics is simple. Borrowing from the design of IP, we pick the network layer as the narrow waist across diverse NFs.

Listing 2: BDU is the I/O unit of an *NF* model.

```
struct BDU{
 // IP fields
 int srcIP, dstIP, proto;
 // transport
 int srcPort,  dstPort;
 // TCP specific
 int tcpSYN, tcpACK, tcpFIN, tcpRST;
 // HTTP specific
 int  httpGetObj, httpRespObj;
 // BUZZ-specific
 int dropped, networkPort, BDUid;
 // Each NF updates traffic context
 int c-Tag[C_TAG_MAX]; //context tags
 int p-Tag; //provenance tag
 ...};
```

Each *NF* model processes only relevant fields of an input BDU (e.g., an L2 switch ignores HTTP fields).

2. *Tag fields for context and provenance:* First, to ensure a BDU carries its context as it goes through the network, we introduce the context tag, or `c-Tag`, field, which explicitly binds the BDU to its context (e.g., 1 bit for cache hit/miss, 1 bit for alarm/no-alarm). When the *NF* model receives an input BDU, it generates an output BDU with the updated `c-Tag` (e.g., a proxy may set the cache hit bit). Second, a BDU preserves its provenance via its `p-Tag` field. This field encodes the BDU's original 5-tuple indicating its *TrafficSpec*. This binding is needed because certain *NFs* (e.g., NATs, proxies) rewrite the original IP 5-tuple of a BDU. We ensure the provenance field `p-Tag` is left unchanged by *NF* models the BDU traverses.
3. *Aggregation for scalability:* Each BDU can represent a sequence of packets associated with higher-layer *NF* operations. This aggregation helps shrink the search space for finding test traffic (§6). For example, all packets of an HTTP reply are captured by a single BDU with the `httpRespObj` field indicating the retrieved object id; a proxy's state (e.g., cache contents) gets updated after receiving this BDU.

To design a BDU struct in practice, we need to identify the protocols that affect any context mentioned in the policies. The struct's fields are simply the union of the policy-related headers of these protocols. For example, if our policy involves a stateful firewall, then TCP `SYN` and `ACK` should be part of the fields, as these are the fields that denote connection establishment semantics. Since each *NF* model processes only relevant fields of an incoming BDU, our BDU abstraction is future-proof. For example, if we later need to add an ICMP field to the BDU of Listing 2, existing *NF* models will remain unchanged, as they simply ignore this new field.

Ensemble of FSMs representation: While there are many ways to expressively model a stateful *NF*, not all models may be scalable. To see why, consider modeling the state-space as the concatenation of state variables we have identified (e.g., in a proxy this concatenation may have three variables: per-host and per-server connection states and per-object cache state). Taking this approach means with *var* variables each with *val* possible values, such a monolithic FSM has val^{var} states (i.e., an exponential growth with the number of values). While it may be tempting to reduce the state space by moving to a layer-specific abstraction (e.g., a proxy model that ignores TCP and purely works at the HTTP layer), this is not viable, as the models of diverse *NFs* will not be composable.

To build a scalable FSM without compromising composability, we borrow insights from the design of actual *NFs*. *NF* programs in practice are not monolithic; rather, they independently track "active" connections, and different functional components of an *NF* are segmented; e.g., client- vs. server-side handling in a proxy are separate. This naturally suggests two opportunities:

1. *Decoupling independent traffic units:* Consider a stateful firewall. If modeled as a monolithic FSM, each state of the model involves states of individual connections. While this is expressive, it is not scalable as the number of connections grow. By decoupling per-connection states, we model the *NF* as an ensemble of FSMs. In general, this insight cuts the number of states from $|state|^{|conn|}$ to $|conn| \times |state|$, where $|conn|$ and $|state|$ denote the number of connections and states per connection, respectively.
2. *Decoupling independent tasks:* To illustrate this, consider a proxy. The code of a real proxy (e.g., `Squid` [18]) typically has three logical modules in charge of managing client-side and server-side connections and the cache. We decouple such logically independent tasks in the model so that instead of a monolithic FSM model with each state being of the "cross-product" form $\langle client\_TCP\_state, server\_TCP\_state, cache\_content\rangle$, we use an *ensemble* of three smaller FSMs, i.e., $\langle client\_TCP\_state\rangle$, $\langle server\_TCP\_state\rangle$, and $\langle cache\_content\rangle$. In general, if an *NF* has $|T|$ independent tasks with task i having S_i states, this idea cuts the number of states from $\prod_{i=1}^{|T|} |S_i|$ to $\sum_{i=1}^{|T|} |S_i|$.

Putting it together: Taken together, our BDU abstraction as the traffic I/O unit and FSM ensembles as *NF* models satisfy the three modeling requirements of composability, expressiveness, and scalability (§5.1). As an illustration, Listing 3 shows a code snippet of a proxy model focusing on the actions when a client requests a non-cached HTTP object and while the proxy has not established a TCP connection with the server. Each *NF* instance is identified by a unique `id` that allows us to index into the relevant variables. Since the traffic I/O of the model (Line 1) is a BDU, the model is com-

Listing 3: Proxy as an ensemble of FSMs.

```
1   BDU  Proxy(NFId id, BDU inBDU){
2       ...
3       if ((frmClnt(inBDU)) && (isHttpRq(inBDU))){
4          if (!cached(id, inBDU)){
5                if(srvConnEstablished(id, inBDU))
6                   outBDU=rqstFrmSrv(id, outBDU);
7                else
8                   outBDU=tcpSYNtoSrv(id, inBDU); }}
9       //set c-Tags based on context (e.g., hit/miss)
10      outBDU.c-Tags = ...
11      ...
12      return outBDU;}
```

posable with other *NF* models. Second, instead of a monolithic FSM, it is partitioned into these three dimensions (i.e., client-, server-side connections and cache) making the model scalable. The state variables of different proxy instances are naturally partitioned per *NF* instance (not shown) and help track the relevant *NF* states, and are updated by the *NF*-specific functions such as `srvConnEstablished`.[3] If the input `inBDU` is an HTTP request (Line 3) and the requested object is not cached (Line 4), the proxy checks the status of the server TCP connection. If it has already been established (Line 5), the output BDU is an HTTP request (Line 6). Otherwise, the proxy initiates a TCP connection with the server (Line 8). Finally, note that the proxy updates `c-Tags` of the output BDU before sending it out.

5.3 Composing the data plane model

Next we discuss how to instantiate a model of the data plane given the models of individual *NFs*. Listing 4 illustrates this for the network of Figure 2. BUZZ uses the policy to automatically concretize the relevant model parameters (e.g., lines 3–4 specify which content/host to watch). Lines 8–10 model the stateless switch, where we model a switch as a static data store lookup [52]. Note that a BDU captures its current location in the network via its `networkPort` field, which gets updated as it traverses the network. Function `lookUp()` takes an input BDU, looks up its forwarding table, and creates a new `outBDU` with its port value set based on the forwarding table.

Similar to prior work [52, 75], our network model processes one-packet-per-*NF* at a time, without modeling (a) batching or queuing inside the network, (b) parallel processing inside *NFs*, or (c) simultaneous processing of different packets across *NFs*. As a result, the data plane model is a simple loop (Line 26); in each iteration, a BDU is processed (Line 27) in two steps: (1) it is forwarded to the other end of the current link (Line 28), (2) it is then passed as an argument to the *NF* connected at this end (e.g., a switch or firewall) (Line 29). The output BDU is then processed in the next iteration. The loop is executed until the BDU is "DONE"; i.e., it either reaches its destination or is dropped by an *NF*.[4] Based on the policy, wee identify the `Next_NF` in line 29. (As an optimization, our implementation prepopulates switches' `lookup()` and `Next_NF()` based on shortest-path routing between policy-relevant *NFs*.) The role of the `assert` statement will become clear in §6, where we discuss test traffic generation.

Listing 4: Data plane pseudocode for Figure 2.

```
1   // Symbolic BDUs to be instantiated (see §6).
2   BDU A[20];
3   int objToWatch = XYZ.com;
4   int hostToWatch = H2;
5   // Global state variables
6   bool Cache[2][100]; // 2 proxies,  100 objects
7   // Model of a switch
8   BDU  Switch(NFId id, BDU inBDU){
9       outBDU=lookUp(id, inBDU);
10      return outBDU;}
11  // Model of a monitoring NF
12  BDU  Mon(NFId id, BDU inBDU){
13       ...
14       outBDU = inBDU;
15       if (isHttp(id, inBDU)){
16         takeMonAction(id, inBDU);/* if inBDU
17         contains objToWatch destined to
18         hostToWatch, set outBDU.dropped to 1.*/}
19       ...
20      return outBDU;}
21  // Model of a proxy NF; See Listing 3
22  BDU Proxy(NFId id, BDU inBDU){...}
23  main(){
24    // Model of the data plane
25    initializeProvenanceTags(A[]);
26    for each injected A[i]
27     while (!DONE(A[i])){
28      Forward A[i] on current link;{
29      A[i] = Next_NF(A[i]);{
30      assert(
31      (!(A[i].p-Tag==hostId[H2]))
32      ||(!(A[i].c-Tags[cacheContext]==objToWatch));
33  }}}}
```

6 Test Traffic Generation

In this section, we discuss how to generate test traffic given the policies and the data plane model. First, we highlight why we choose symbolic execution (SE) as a starting mechanism to explore the data plane model (§6.1). Then we present our domain-specific optimizations to scale SE to generate abstract test traffic consisting of BDUs (§6.2). Then, we show how to convert this abstract test traffic into concrete test traffic (§6.3). Finally, we present an extension to test dynamic *NF* scenarios (§6.4).

6.1 Why symbolic execution (SE)?

For BUZZ to be usable by operators at human interactive timescales, it should generate test traffic within seconds to a few minutes even for large networks. This is challenging on two fronts:

[3] The choice of passing *id*s and modeling state in per-*id* global variables is not fundamental but an artifact of using C/KLEE.

[4] *NFs* may be time-triggered (e.g., TCP time-out), so we capture time using a BDU field. These "time BDUs" are injected by the network model periodically to update time-related states.

- **Traffic space explosion:** Unlike prior work where an IP packet header is an independent unit of test (hence mandating a search only over the header space [51, 53, 75, 76]), we need to search over a very large *traffic space* of all possible sequences of traffic units. While BDUs, as compared to IP packets, improve scalability via aggregation (§5.2), we still have to search over the space of possible BDU value assignments.
- **State space explosion:** Even though using the FSM ensembles abstraction significantly reduces the number of states (§5.2), it does not address *state space explosion* due to composition of *NFs*; e.g., if the models of NF_1 and NF_2 can reach K_1 and K_2 states, respectively, their composition will have $K_1 \times K_2$ states.

Unfortunately, several canonical search solutions (e.g., model checking [4, 36] and AI planning tools [7]) do not scale beyond 5-10 stateful *NFs*; e.g., model checking took 25 hours for policy involving only two contexts.

As the first measure to address the search scalability challenge, we choose symbolic execution (SE), which is a well-known approach to tackle state-space explosion [30]. At a high level, an SE engine explores possible behaviors of a program (in our case, the data plane model) by assigning different values to its *symbolic variables* [32]. In our implementation, we use KLEE [31], a popular SE engine.

6.2 Generating abstract test traffic

BUZZ employs SE as follows. For each $policy : trafficSpec \times contextSeq \mapsto portSeq$, we constrain the symbolic BDUs to satisfy the *TrafficSpec*. Then, to drive the SE engine to generate test traffic that satisfies $contextSeq = \langle context_{NF_1}, \dots, context_{NF_N} \rangle$, we introduce the logical negation of *contextSeq* as an *assertion* in the network model code. In practice, if *contextSeq* involves contexts of N *NFs* $context_1, \dots, context_N$, BUZZ instruments the network model with an assertion of the form $\neg(context_1 \wedge \dots \wedge context_N)$, where each term is expressed in terms of BDUs' `c-Tag` sub-fields. The assertion guides the SE engine toward finding a "violation" of the assertion by assigning concrete values to symbolic BDUs.[5] In effect, SE generates *abstract test traffic* by concretizing a sequence of symbolic BDUs. The abstract test traffic will be then translated into concrete test traffic (§6.3), which in turn, will be injected into the actual data plane. The injected concrete test traffic must traverse the sequence of ports specified in *portSeq*; otherwise, the actual data plane violates *policy*.

To illustrate this, let us revisit Listing 4, where we want a test trace to check cached responses from the proxy to host H_2. Lines 30-32 show the assertion to get a sequence of i BDUs that change the state of the data plane such that the ith BDU in the abstract traffic trace: (1) is from host H_2 (Line 31), and (2) corresponds to a cached response (Line 32). For example, the SE engine may generate 6 BDUs: three BDUs between a host other than H_2 in the *Dept* and the proxy to establish a TCP connection (the 3-way handshake) where the third BDU has `httpGetObj = httpObjId` (this effectively makes the proxy cache the object), followed by another 3 BDUs, this time from H_2 with the field `httpGetObj` set to `httpObjId` to induce a cached response. Similarly, Listing 5 shows an assertion in Lines 7-8 to trigger alarms at both L-IPS and H-IPS of the example from Figure 3.

Listing 5: Assertion pseudocode for Figure 3 to trigger alarms at both IPSes.

```
1 // Global state variables
2 int L_IPS_Alarm[noOfHosts];//alarm per host
3 int H_IPS_Alarm[noOfHosts];//alarm per host
4 ...
5 //A[] is an array of symbolic BDUs
6 ...
7     assert((!(A[i].c-Tags[L_IPS_Alarm]==1)) ||
8            (!(A[i].c-Tags[H_IPS_Alarm]==1)));
```

While SE is significantly faster than other candidates, it is not sufficient for interactive use. Even after a broad sweep of configuration parameters to customize KLEE, it took several hours for a small network (§8.3). To scale to large topologies, we implement two optimizations:

- **Minimizing number of symbolic variables:** Making an entire BDU structure (Listing 2) symbolic forces KLEE to find values for every field. Instead, BUZZ identifies the policy-related subset of BDU fields and only makes these symbolic and concretizes the rest. For instance; when BUZZ is testing a data plane with a stateful firewall but no proxies, it makes the HTTP-relevant fields concrete (i.e., non-symbolic) by assigning a don't care value $*$ (represented by -1 in our implementation) to them.
- **Scoping values of symbolic variables:** The *trafficSpec* scopes the range of values a BDU may take. BUZZ further narrows this range using the policy and protocols semantics. For example, even though the `tcpSYN` field is an integer, BUZZ constrain it to be either 0 or 1.

Test coverage: Ideally, test traffic should cover the space of all possible traffic, including (1) packet traces of all possible lengths (in terms of number of packets in the trace), and (2) enumerating all possible values of the fields of each packet. However, this is impractical with respect to both test traffic generation and injection overheads. That is why even in case of simple reachability policies and stateless data planes in prior work [75], only one sample packet out of an equivalence class of packets (i.e., the set of all packets that experience the same forwarding behavior) is selected as the test packet. Sim-

[5] Note that an assertion of the form $\neg(A_1 \wedge \dots \wedge A_n)$, or equivalently $(\neg A_1 \vee \dots \vee \neg A_n)$, is violated only if each term A_i is evaluated to `true`.

ilarly, we define our test coverage goal as obtaining one test trace to exercise each policy. In §8, we will show that BUZZ (1) successfully satisfies this goal, and (2) can be used to satisfy alternative coverage goals.

6.3 Generating concrete test traffic

The output of the SE step is a sequence of BDUs $BDUSeq^{SE}$. Since BDUs are abstract, we cannot directly inject them into the actual data plane. Moreover, we cannot simply do a one-to-one translation between BDUs and raw packets and do a trace replay [3, 75] because we need to honor session semantics (e.g., for TCP or FTP) of the policies—several parameters of such sessions (e.g., TCP seq. numbers) are outside of our control and are chosen by the OS of the end hosts at run time.

To this end, we translate abstract test traffic into *test traffic injection scripts* that are run on end hosts to inject concrete test traffic. The translation algorithm uses a library of traffic injection commands that maps a known $BDUSeq_l$ into a script. For example, if a BDUSeq consists of 3 BDUs for TCP connection establishment and a web request, we map this into a `wget` with the required parameters (e.g., server IP and object URL). In the most basic case, the script will be an IP packet. Using our domain knowledge, we populated this library with commands (e.g., `getHTTP(.)`, `sendIPPacket(.)`) that support IP, TCP, UDP, FTP, and HTTP.

For completeness, its pseudocode is presented in Appendix A. Here we give the intuition behind our translation algorithm. We partition the $BDUSeq^{SE}$ based on srcIP-dstIP pairs (i.e., communication end-points) of BDUs; i.e., $BDUSeq^{SE} = \bigcup_l BDUSeq_l$. Then for each partition $BDUSeq_l$, we do a longest-specific match (i.e., match on a protocol at the highest possible layer of the network stack) in our test script library, retrieve the corresponding command for each subsequence, and then concatenate these commands to form a traffic injection script.

6.4 Testing dynamic NF deployments

Next we describe the extensions needed to handle dynamic NF deployment scenarios. Intuitively, the goal in these scenarios is to ensure the change is transparent with respect to stateful semantics of traffic. To be concrete, let $Policy_{before}$ and $Policy_{after}$ denote the policies that the operator intends to enforce before and after the “change” occurs, where the change is captured by *changeCond* (e.g., an NF's scale-out, or failure). We define the correct enforcement of a dynamic NF deployment policy as follows: For each data plane state $s \in S_{DP}$, if *changeCond* is triggered while the data plane is in s, then $Policy_{after}$ is enforced correctly.

In Figure 4, $Policy_{before}$ is the top part of the policy graph (i.e., involving IPS_1), $Policy_{after}$ is the bottom part of the policy graph (i.e., involving IPS_2), and *changeCond* is IPS_1's scale-out. Irrespective of the state in which IPS_1 scales out, IPS_2 must start processing traffic with the same state at which IPS_1 has scaled out.

Abstract test traffic generation for dynamic NF deployment scenarios is slightly different from what we described in §6.1. At a high-level, for every data plane state $s \in S_{DP}$, BUZZ (1) generates test traffic to drive the data plane to s, (2) triggers *changeCond* (e.g., by scaling-out an NF), and (3) test if the data plane is compliant with $Policy_{after}$. For completeness, we describe the full procedure in Appendix B.

7 Implementation

BUZZ comprises ≈ 10,000 lines of code, including NF models, code for test traffic generation, test resolution, extensions to KLEE, and the operator interfaces. The entire workflow of BUZZ is implemented atop `OpenDayLight` [14]. The source code is available at [1].

Operator interface: Operators can enter policies using either a text-based or a graphical interface (example screenshots in Appendix C). BUZZ then performs a set of sanity checks on the policies and warns the operator of any mistakes (e.g., an overlap between *TrafficSpec* of two policies). This I/O is the only effort that BUZZ needs from the operator. Once policies are entered, the workflow of BUZZ (Figure 5) is entirely automated.

NF models: We have written `C` models for switches, ACL devices, stateful firewalls, NATs, L4 load balancers, HTTP and FTP proxies, passive monitoring, and simple intrusion prevention systems (e.g., counting failed connection attempts and matching payload signatures). Our models are between 10 (for a switch) to 100 lines (for a proxy cache) of C code. We reuse common templates across NFs; e.g., TCP connection sequence used in both the firewall and proxy models. Note that modeling NFs is a one-time offline task and can be augmented with community efforts [12]. We validated models by inspecting call graphs visualization [9, 21] on extensive manually generated input traffic to ensure the models are correct.

Test traffic generation and injection: We use KLEE with the optimizations discussed in §6.2 to generate BDU-level test traffic (i.e., abstract test traffic), and then translate it to test scripts that run at the injection points.

Test traffic monitoring and test resolution: We use offline monitoring via `tcpdump` (with suitable filters). BUZZ uses the monitoring logs to determine the test result. For completeness, we have provided the monitoring and test resolution pseudocode in Appendix D. Here we give the intuition behind this process. From the input policy, BUZZ inspects the monitoring logs to check whether traffic has traversed the policy-mandated ports. If so, the test concludes with success. Otherwise, a policy violation along with the first violating port on which traffic appeared is declared.

8 Evaluation

In this section, we show that:

1. BUZZ can help detect a broad spectrum of both new and known policy violations (§8.1);
2. BUZZ works in close-to-interactive time scales (i.e., within two minutes) even for large topologies with 100s of switches and stateful *NFs* (§8.2); and
3. BUZZ's design is critical for its scalability (§8.3).

Testbed and topologies: We use a testbed of 13 server-grade machines (20-core 2.8GHz servers with 128GB RAM) connected via direct 1GbE links and a 10GbE Pica8 OpenFlow switch. On each server, with KVM installed, we run injectors and software *NFs* as separate VMs, connected via `Open vSwitch`. The specific stateful *NFs* are iptables [8] as a NAT and a stateful firewall, Squid [18] as a proxy, Snort [17] and Bro [65] as IPS/IDS, Balance [2], and PRADS [15].

In addition to the example scenarios from §2, we use 8 randomly selected recent topologies from the Internet Topology Zoo [19] with 6–196 nodes. We also use two larger topologies (400 and 600 nodes) by extending these topologies. These serve as switch-level topologies; we extend them with different *NFs* to enforce policies. For the scalability experiments, we augment each switch-level topology with stateful *NFs* (§8.2) by connecting each stateful *NF* to a randomly selected switch. As a concrete policy enforcement scheme, we used prior work to handle dynamic middleboxes [43]. (We reiterate designing this scheme is not the goal of BUZZ; we simply needed *some* concrete solution.)

8.1 BUZZ end-to-end use cases

First, we demonstrate the effectiveness of BUZZ in finding both new and known policy violations.

Finding new violations: Using BUZZ, we uncovered several policy violations in recent systems, a few of which we present here:

- **Violations due to reactive control in Kinetic [10]**: We set up a simple policy composed of an IDS followed by a Kinetic dynamic firewall. By generating malicious traffic, BUZZ found that the first few malicious packets are wrongly let through. The root cause of this violation is the delay between (1) the IDS's detection of malicious traffic and sending an "infected" event to the controller, and (2) the controller's reconfiguration of the data plane to block malicious traffic.
- **Incorrect state migration using OpenNF [46]**: We used the OpenNF-enhanced PRADS [15, 46] to enforce the following policy: if a host spawns more than *Thresh* TCP connections, its traffic should be sent to a rate limiter. BUZZ revealed a violation due to the incorrect state migration when we elastically scale a PRADS instance. Specifically, BUZZ made a host establish n_1 and n_2 sessions with a server before and after migration, respectively, such that: n_1, $n_2 < Thresh$, but $n_1 + n_2 > Thresh$. BUZZ then found that traffic did not go to the rate limiter. This is because OpenNF does not migrate the session count (i.e., n_1) from $PRADS_1$ to $PRADS_2$.
- **Faulty policy composition using PGA [66]**: We used PGA[6] to compose two policies on traffic from H_1 to H_2: it should pass a load balancer and a stateful firewall ($policy_1$), and if it is found suspicious, it then should go to an IPS ($policy_2$). After enforcing the composition of the two policies, BUZZ found that the test traffic exercising $policy_1$ did not go through the firewall. This is because the SDN switch rules corresponding to $policy_1$ took precedence over the switch rules for $policy_2$, rendering $policy_2$ ineffective.
- **Incorrect tagging using FlowTags [43]:** BUZZ helped us identify a bug in our FlowTags implementation in OpenDaylight [14]. In the scenario of Figure 2, the controller code in charge of decoding tags (e.g., to distinguish hosts behind the proxy) would assign the same tag value to traffic from different hosts. Our test traffic showed that the proxy's cache hit replies bypass the monitoring device. BUZZ's traffic trace indicated that the tag values of cache miss/hit are identical; this gave us a hint as to focus on the controller code in charge of configuring the tagging behavior of the proxy.

Finding known violations: We used a "red team–blue team" exercise, to evaluate the utility of BUZZ in finding known policy violations. In each scenario, the red team (Student 1) secretly picks one of the policies (at random) from the set of policies that is known to both teams, and creates a failure that causes the network to violate this policy; e.g., misconfiguring L-IPS count threshold. The blue team (Student 2) uses BUZZ to identify a violation and localize the source of the policy violation.

Table 1 highlights the results for a subset of these scenarios and the specific traces that BUZZ generated. Three of the scenarios use the motivating examples from §2. In the Conn. limit. scenario, two hosts are connected to a server through an authentication server to prevent brute-force password guessing attacks. The authentication server is expected to halt a host's access after 3 consecutive failed log in attempts. Finally, in the asymmetric routing scenario, upstream and downstream traffic traverse different paths [55]. In all scenarios, the blue-team

[6] We used our implementation of PGA, as its code was unavailable.

“Red Team” scenario	BUZZ test trace	Violating NF
Cascaded NATs using Click IPRewriter [54] ; NAT_2 incorrectly rewrites srcIP triggering “assertion failure” on NAT_1 [38]	H_1 attempts to access to the server	NAT_2
Multi-stage triggers (Fig. 3); L-IPS miscounts by summing three hosts	H_1 makes 9 scan attempts followed by 9 scans by H_2	L-IPS
Conn. limit.; Login counter resets	H_1 makes 3 continuous log in attempts with a wrong password	Login counter
Conn. limit.; S_1 missing switch forwarding rules from AuthServer to the protected server	H_2 makes a log in attempt with the correct password	S_1
Conflicting firewall rules: Rule 1, if internal connect to external IP, allow IP to access any internal port; Rule 2, block external access to internal port 443	A tcp connection from internal C_1 to external S_1 followed by an access from S_1 to $C_1 : port443$	Firewall
Asymmetric routing; Client-to-server TCP traffic goes through Bro, but the response bypasses Bro. Since Bro does not see the SYN_ACK packet, it (mistakenly) blocks the connection.	a tcp connection followed by tcp data packets	switch close to dst.

Table 1: Example red-blue team scenarios.

successfully localized the failure (i.e., which NF or link is the root cause) within 10 seconds.

It is useful at this time to reiterate that these types of violations could not be exposed by existing debugging tools such as ATPG [75], ping, or traceroute, as they do not capture violations w.r.t. stateful/context-dependent aspects. We also tried using fuzzing to generate test traffic, using both Scapy [16] and a custom fuzzer. Across all scenarios, fuzzing did not find any test trace within 48 hours. This is because we need targeted search to trigger specific data plane states, which fuzzing is not suited for.

8.2 Scalability

Recall that we envision operators using BUZZ in an interactive fashion; i.e., the time for test generation should be within 1-2 minutes even for large networks with hundreds of switches and stateful NFs.

We evaluate how BUZZ scales with topology size and policy complexity. We define policy complexity as the number of stateful NFs whose contexts appear in the policy. We consider a baseline policy that has 3 stateful NFs (a NAT, followed by a proxy, followed by a stateful firewall). The firewall is expected to block access from a fixed subset of origin hosts to certain web content. To create more complex policies, we linearly “chain” together repetitions of the baseline policy.

Figure 7 shows the average test traffic generation latency for various topology sizes and policy complexities. There are two takeaways. First, BUZZ generates test traffic in human-interactive time scales; even in the largest topology with 600 switches and the most complex policy it takes only 113 seconds. Second, BUZZ’s test traffic generation latency only depends on the policy complexity: if we increase the topology size without increase the policy complexity, this will not add to the test traffic generation latency. This is expected, as test traffic generation involves a search over the data plane state space, which naturally is a function of stateful NFs.

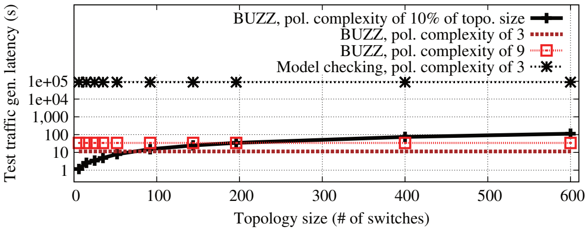


Figure 7: Test generation latency of BUZZ.

To put the traffic generation latency of BUZZ in perspective, Figure 7 also shows the traffic generation latency of a strawman solution of using the model checker CMBC [4]. Even on a small 9-node topology (6 switches and 3 stateful NFs), it took 25 hours; i.e., on a 90× larger topology, BUZZ is *at least* five orders of faster.

Test coverage: We have evaluated the test coverage of BUZZ, and here, we discuss the three takeaways. First, across all scenarios of §8.1 and §8.2, we explicitly enumerated all contexts, and observed that BUZZ provided full coverage with respect to the coverage goal of §6.1 (i.e., one test case to trigger each context). Second, we extended BUZZ to satisfy an alternative coverage goal of generating > 1 test trace per context. We enabled this through an iterative test generation process, where in each iteration, we obtain a new test case by using assertions such that a previously generated test case will not be generated again. Finally, while, in general, using multiple test cases per context may reveal new violations, in our experiments, we did not find new violations by doing so.

8.3 BUZZ design choices

Next, we do a component-wise analysis to demonstrate the effect of our key design choices and optimizations.

BDUs vs. packets: To see how aggregating a sequence of packets as a BDU helps with scalability, we use BUZZ to generate test traffic to test the proxy-monitor policy (Figure 2), first in terms of BDUs and then in terms of raw MTU-sized packets, on varying sizes of files to retrieve from the web. Figure 9 shows that on the topology with 600 switches and 300 stateful NFs, in case of packet-level test traffic generation, test traffic generation latency increases linearly with the file size. On the other hand, since the number of test packets is dominated by the number of object retrieval packets, aggregating all file retrieval packets as one BDU significantly cuts the latency. (The results, not shown, are consistent across topologies as well as using FTP instead of HTTP.)

Impact of SE optimizations: We examine the effect of the SE-specific optimizations (§6.2) in Figure 8. To put

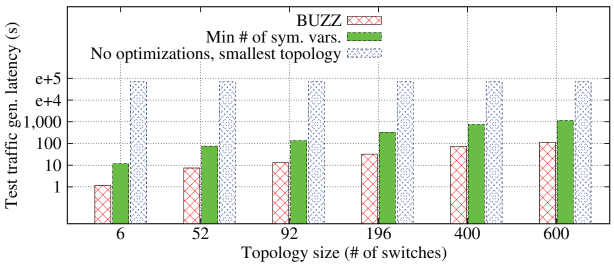


Figure 8: Improvements due to SE optimizations.

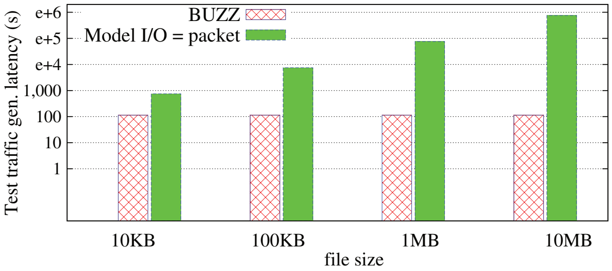


Figure 9: BDUs vs. packets for various request sizes.

these numbers in context, using KLEE without the optimizations on a network of six switches and a policy chain with three stateful NFs takes $\geq$ 19 hours. We see that (1) minimizing the number of symbolic variables cuts the test generation latency by three orders of magnitude, and (2) scoping the values yields a further $> 9\times$ reduction.

9 Related work

Network verification: There is a rich literature on checking reachability [40, 44, 51, 52, 57, 58, 73, 74]. The work closest to BUZZ is ATPG [75]. As discussed earlier, these do not capture the stateful behaviors and context-dependent policies.

Code verification: The work in [39] focuses on finding Click [54] code faults (e.g., crash) as opposed to verifying traffic processing policies (e.g., reachability). NICE combines model checking and SE to find bugs in control plane software [33]. BUZZ is complementary to these efforts.

Modeling stateful networks: Joseph and Stoica formalized middlebox forwarding behaviors but do not model stateful behaviors [50]. The only work that also models stateful behaviors are FlowTest [41], Symnet [71], and the work by Panda et al [64]. FlowTest's [41] high-level models are not composable and the AI planning approaches do not scale beyond 4-5 node networks. Symnet [71] uses models written in Haskell to capture NAT semantics similar to our example; based on published work we do not have details on their models, verification procedures, or scalability. The work by Panda et al. is different from BUZZ in terms of both goals (only reachability policies) and techniques (static checking) [64].

Policy enforcement: There are several frameworks to facilitate policy enforcement [10,43,46,63,66,67]. There are also efforts to generate correct-by-construction SDN programs [25, 27, 45]. Our work is complementary, as it checks whether the intended behavior manifests correctly in the actual data plane.

Simulation and shadow configurations: Simulation [13], emulation [6, 11], and shadow configurations [24] are common methods to model/test networks. BUZZ is orthogonal in that it focuses on generating test traffic. While our current focus is on active testing, BUZZ applies to these platforms as well. We also posit that our techniques can be used to validate these efforts.

10 Discussion

Model synthesis: BUZZ uses hand-generated models of NFs. A natural direction for future work is to use program analysis to automatically synthesize NF models from middlebox code (e.g., [35]) or logs (e.g., [28]).

Soundness vs. completeness: For "infinite-state" systems, it is not possible to simultaneously achieve both guarantees [49]. BUZZ's design favors soundness (i.e., if we report a violation, then the data plane actually has that behavior) over completeness (i.e., if we do not find a violation, then there are no bugs). In our setting, this is a worthwhile trade-off as we can repeat tests for greater coverage [49, 75] (e.g., see §8.2).

New use cases: Looking forward, we believe BUZZ can be extended to systematically check interoperability of new protocols with middleboxes [48]. As preliminary evidence, we were able to replicate a known problem with a middlebox-cooperative TCP extension called HICCUPS [37], where the protocol fails in the presence of middleboxes that modify certain headers or if there are multiple middleboxes on the path.

11 Conclusions

BUZZ tackles a key missing piece in network verification—checking *context-dependent policies* in *stateful data planes* introduces fundamental expressiveness and scalability challenges. We make two key contributions to address these challenges: *(1)* Developing expressive and scalable network models; and *(2)* An optimized application of symbolic execution to tackle state-space explosion. We demonstrate that BUZZ is scalable and it can help diagnose policy violations.

Acknowledgments

This work was supported in part by grant number N00014-13-1-0048 from the Office of Naval Research, NSF awards 1440056 and 1440065, and Intel Labs' University Research Office. Seyed K. Fayaz was supported by the VMware Graduate Fellowship and CMU Bertucci Fellowship. We thank the anonymous reviewers and our shepherd Kobus Van der Merwe for their suggestions.

References

[1] BUZZ. https://github.com/network-policy-tester/buzz.

[2] Balance. http://www.inlab.de/balance.html.

[3] Bit-Twist. http://bittwist.sourceforge.net.

[4] CBMC. http://www.cprover.org/cbmc/.

[5] Cisco's Reflexive Access Lists. http://bit.ly/1O8N5p2.

[6] Emulab. http://www.emulab.net/.

[7] Graphplan. http://www.cs.cmu.edu/~avrim/graphplan.html.

[8] iptables. http://www.netfilter.org/projects/iptables/.

[9] KCachegrind. http://kcachegrind.sourceforge.net/html/Home.html.

[10] Kinetic. http://resonance.noise.gatech.edu/.

[11] Mininet. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

[12] Network Function Virtualization Research Group (NFVRG).

[13] ns-3. http://www.nsnam.org/.

[14] OpenDaylight project. http://www.opendaylight.org/.

[15] PRADS. http://gamelinux.github.io/prads/.

[16] Scapy. http://bit.ly/1FiqZyK.

[17] Snort. http://www.snort.org/.

[18] Squid. http://www.squid-cache.org/.

[19] The Internet Topology Zoo. http://www.topology-zoo.org/index.html.

[20] Troubleshooting the network survey. http://eastzone.github.io/atpg/docs/NetDebugSurvey.pdf.

[21] Valgrind. http://www.valgrind.org/.

[22] High Performance Service Chaining for Advanced Software-Defined Networking (SDN) . http://intel.ly/1ilX5PG, 2014.

[23] Tackling the Dynamic Service Chaining Challenge of NFV/SDN Networks with Wind River and Intel. http://intel.ly/1EFmEVQ, 2014.

[24] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. In *Proc. SIGCOMM*, 2008.

[25] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.

[26] B. Anwer, T. Benson, N. Feamster, D. Levin, and J. Rexford. A slick control plane for network middleboxes. In *Proc. HotSDN*, 2013.

[27] T. Ball, N. Bjorner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarskyi. VeriCon: Towards verifying controller programs in software-defined networks. In *Proc. PLDI*, 2014.

[28] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proc. ESEC/FSE*, 2011.

[29] M. Boucadair, C. Jacquenet, R. Parker, D. Lopez, J. Guichard, and C. Pignataro. Differentiated Service Function Chaining Framework. https://tools.ietf.org/html/draft-boucadair-service-chaining-framework-00, 2013.

[30] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10ˆ20 States and Beyond. *Inf. Comput.*, 98(2), 1992.

[31] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, 2008.

[32] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[33] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test openflow applications. In *Proc. NSDI*, 2012.

[34] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng, J. Benitez, U. Michel, H. Damker, K. Ogaki, T. Matsuzaki, M. Fukui, K. Shimano, , D. Delisle, Q. Loudier, C. Kolias, I. Guardini, E. Demaria, R. Minerva, A. Manzalini, D. Lpez, F. Javier, R. alguero, F. Ruhl, and P. Sen. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. http://portal.etsi.org/nfv/nfv_white_paper.pdf, 2012.

[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855. 2000.

[36] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[37] R. Craven, R. Beverly, and M. Allman. A middlebox-cooperative tcp for a non end-to-end internet. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 151–162. ACM, 2014.

[38] M. Dobrescu, K. Argyarki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *Proc. NSDI*, 2012.

[39] M. Dobrescu and K. Argyraki. Software dataplane verification. In *Proc. NSDI*, 2014.

[40] D. J. Dougherty, T. Nelson, C. Barratt, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *Proc. LISA*, 2010.

[41] S. K. Fayaz and V. Sekar. Testing stateful and dynamic data planes with FlowTest. In *Proc. HotSDN*, 2014.

[42] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic DDoS defense. In *Proc. USENIX Security Symposium*, 2015.

[43] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *Proc. NSDI*, 2014.

[44] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Proc. NSDI*, 2015.

[45] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. *SIGPLAN Not.*, 46(9), Sept. 2011.

[46] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *Proc. SIGCOMM*, 2014.

[47] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *ACM Queue*, 2012.

[48] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? In *Proc. IMC*, 2011.

[49] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 2009.

[50] D. Joseph and I. Stoica. Modeling middleboxes. *Netwrk. Mag. of Global Internetwkg.*, 22(5), 2008.

[51] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. NSDI*, 2013.

[52] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proc. NSDI*, 2012.

[53] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.

[54] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 2000.

[55] F. Le, E. Nahum, V. Pappas, M. Touma, and D. Verma. Experiences deploying a transparent split tcp middlebox and the implications for nfv. In *Proc. HotMiddlebox*, 2015.

[56] W. Liu, H. Li, O. Huang, M. Boucadair, N. Leymann, Z. Cao, and J. Hu. Service Function Chaining (SFC) Use Cases. `http://bit.ly/1JTVneh`, 2014.

[57] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *Proc. NSDI*, 2015.

[58] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proc. SIGCOMM*, 2011.

[59] N. McKeown. Mind the Gap: SIGCOMM'12 Keynote. `http://bit.ly/1izyVld`.

[60] N. McKeown et al. OpenFlow: enabling innovation in campus networks. *CCR*, March 2008.

[61] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 1990.

[62] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proc. NSDI*, 2013.

[63] S. Palkar, C. Lan, S. Han, K. J. amd Aurojit Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A framework for NFV applications. In *Proc. SOSP*, 2015.

[64] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying Isolation Properties in the Presence of Middleboxes. `arXiv:submit/1075591`.

[65] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1999.

[66] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. PGA: Using graphs to express and automatically reconcile network policies. In *Proc. SIGCOMM*, 2015.

[67] Z. Qazi, C. Tu, L. Chiang, R. Miao, and M. Yu. SIMPLE-fying middlebox policy enforcement using sdn. In *Proc. SIGCOMM*, 2013.

[68] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Proc. NSDI*, 2013.

[69] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback recovery for middleboxes. In *Proc. SIGCOMM*, 2015.

[70] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. SIGCOMM*, SIGCOMM, 2012.

[71] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Static checking for stateful networks. In *Proc. HotMiddlebox*, 2013.

[72] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5), 2012.

[73] G. Xie, J. Zhan, D. Maltx, H. Z. G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In *Proc. INFOCOM*, 2005.

[74] L. Yuan and H. Chen. FIREMAN: a toolkit for FIREwall Modeling and ANalysis. In *Proc. IEEE Symposium on Security and Privacy*, 2006.

[75] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT*, 2012.

[76] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proc. NSDI*, 2014.

A Translating abstract test traffic into test traffic injection scripts

Figure 10 shows the pseudocode for the translation mechanism (§6.3).

```
▷ Inputs:
#1: a sequence of BDUs from Symbolic Execution
$BDUseq^{SE} = \langle BDU_n : n = 1,2,...,N \rangle$,
each $BDU_n$ has an abstract $pred_d$
#2: a cmd-BDUs library $cmdlib = \{\langle cmd_1, Seq^{cmd}{}_1 \rangle, \langle cmd_2, Seq^{cmd}{}_2 \rangle, ..., \langle cmd_M, Seq^{cmd}{}_M \rangle\}$
#3: a set of end-hosts $H = \{H_k : k = 1,2,...,K\}$ to execute cmds
▷ Outputs:
#1: a number of scripts $S = \{script^{H_1} \cdots script^{H_K}\}$ to
be executed on end-hosts $\{H_k : k = 1,2,...,K\}$,
where $script^{H_k}$ is a sequence of $\langle \cdots cmd_i^{H_k} \cdots \rangle$, such that
$\langle \cdots Seq^{cmd}{}_i^{H_1} \cdots \rangle$ is equivalent to $BDUseq^{SE}$
▷ Sort cmd-BDUs library from cmds with most BDUs to least BDUs
$cmdlib = Sort(cmdlib)$
▷ Decompose $BDUseq^{SE}$ sequence into subsequences $BDUsubseq^{SE}$ of BDUs with same predicate $pred$
$\{BDUsubseq^{SE}{}_{pred_d} : d = 1,2,...,D\} = Decompose(BDUseq^{SE})$
for each $BDUsubseq^{SE}{}_{pred_d}$ in $\{BDUsubseq^{SE}{}_{pred_d} : d = 1,2,...,D\}$
    ▷ Instantiate a $script_{pred_d}$ to store $cmd$ for $BDUsubseq^{SE}{}_{pred_d}$
    $script_{pred_d} \leftarrow empty$
    ▷ Match the BDUs in $BDUsubseq^{SE}{}_{pred_d}$ with $cmds$ in $cmdlib$
    for each $cmd_m$ in $cmdlib$
        for $BDU_n$ in $BDUsubseq^{SE}{}_{pred_d}$
            ▷ if $Seq^{cmd}{}_m$ equals to a BDU substring of $BDUsubseq^{SE}{}_{pred_d}$ started at $BDU_n$
            if $Substring(BDUsubseq^{SE}{}_{pred_d}, BDU_n, len(Seq^{cmd}{}_m)) == Seq^{cmd}{}_m$
                ▷ add the matched $cmd_m$ and the first matched $BDU$'s index $n$ to $script_{pred_d}$
                $script_{pred_d}.add(cmd_m^n)$
                ▷ mark all BDUs in $Substring(BDUsubseq^{SE}{}_{pred_d}, BDU_n, len(Seq^{cmd}{}_m))$
                $Mark(Substring(BDUsubseq^{SE}{}_{pred_d}, BDU_n, len(Seq^{cmd}{}_m)))$
        ▷ remove all marked BDUs from $BDUsubseq^{SE}{}_{pred_d}$
        $RemoveMarked(BDUsubseq^{SE}{}_{pred_d})$
        if all BDUs in $BDUsubseq^{SE}{}_{pred_d}$ are marked, then break
    ▷ Sort every $cmd_m^n$ in $script_{pred_d}$ by its first matched $BDU$'s index $n$
    $script_{pred_d} = Sort(script_{pred_d})$
    ▷ Map abstract $pred_d$ to real test host $H_{pred_d}$ and assign script to host
    $script_{H_{pred_d}} = script_{pred_d}$
```

Figure 10: Pseudocode for translating abstract test traffic into test traffic injection scripts.

B Abstract test traffic generation for change management policies

Figure 11 shows the abstract test traffic generation pseudocode for change management policies (§6.4).

C Operator interface of BUZZ

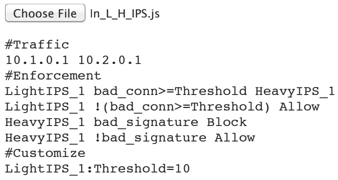
Figures 12 and 13 show operator's interface (§7).

D Test resolution

Figure 14 shows the pseudocode for test resolution (§7).

```
▷ Inputs:
#1: $Policy_1 : pred_1(5{-}tuple) \times C_1 \mapsto Ports_1$ before migrate/rollback
#2: $Policy_2 : pred_2(5{-}tuple) \times C_2 \mapsto Ports_2$ after migrate/rollback
▷ Outputs:
#1: a sequence of $BDUseq^{SE} = \langle BDU_n : n = 1,2,...,N \rangle$ with two substrings,
$BDUseq^{SE}{}_{before}$ and $BDUseq^{SE}{}_{after}$, which should satisfy:
$BDUseq^{SE}{}_{before}$ exploits all possible context $context$ in $C_1$ before migration/rollback happens.
$BDUseq^{SE}{}_{after}$ test all possible $context$ in $C_2$ after the migration/rollback.
▷ Init BDU sequence
$BDUseq^{SE} = \langle BDU_n : n = 1,2,...,N \rangle$
▷ note the values in $BDU_n$ for calculation by Symbolic Execution
$makesymbolic(BDU_n)$
▷ exploits all possible $context$ in $C_1$
▷ $BDUs$ processed sequentially by $Policy_1$
for each $BDU_i$ in $BDUseq^{SE}$
    if $BDU_i$ is in $BDUseq^{SE}{}_{before}$
        ▷ process $BDU_i$ by $Policy_1$ and update $C_1$
        $C_1 = Policy_1(pred_1(BDU_i), C_1, Ports_1)$
▷ do migrate/rollback and change service chain from $Policy_1$ to $Policy_2$
▷ map ports
$Ports_2 = g(Ports_1)$
▷ migrate/rollback context
$C_2 = C_1$
▷ test all possible $context$ in $C_2$
▷ $BDUs$ processed sequentially by $Policy_2$
for each $BDU_j$ in $BDUseq^{SE}$
    if $BDU_j$ is in $BDUseq^{SE}{}_{after}$
        ▷ process $BDU_i$ by $Policy_2$ and update $C_2$
        $C_2 = Policy_2(pred_2(BDU_i), C_2, Ports_2)$
▷ generate $BDU$ sequence with values assigned by Symbolic Execution
$symbolicoutput = \langle BDU_n : n = 1,2,...,N \rangle$
```

Figure 11: Pseudocode for abstract test traffic generation for change management policies.



```
#Traffic
10.1.0.1 10.2.0.1
#Enforcement
LightIPS_1 bad_conn>=Threshold HeavyIPS_1
LightIPS_1 !(bad_conn>=Threshold) Allow
HeavyIPS_1 bad_signature Block
HeavyIPS_1 !bad_signature Allow
#Customize
LightIPS_1:Threshold=10
```

Figure 12: Text-based interface to input policies (e.g., multistage-triggers policy in Figure 3).

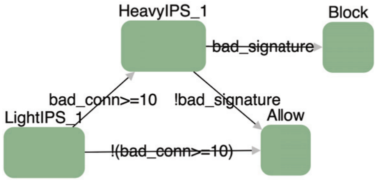


Figure 13: Graphical interface to input policies (e.g., multistage-triggers policy in Figure 3).

```
▷ Inputs:
#1: packet traces $pkttrace_{port_i}$ dumped at each $port_i$ in $Ports$
#2: policy $Policy : pred(5{-}tuple) \times C \mapsto Ports$, where $C$ includes all possible contexts
▷ Outputs:
#1: The resolution result of each context $context_i$ in $C$ in terms of pass/fail
#2: The port of the NF that causes the failure
▷ perform resolution scheme for each context $context_i$ in $C$
for each $context_i$ in $C$
    ▷ $Trace = \langle pkt_m, \ldots, pkt_r \rangle$ is the test packets for this context
    for testpkt in $\langle pkt_m, \ldots, pkt_r \rangle$
        ▷ calculate the logically correct ports $testpkt$ should reach
        $Ports_{testpkt}^{logical} = Policy(pred(testpkt)), context_i)$
        ▷ find the real ports $testpkt$ has reached
        $Ports_{testpkt}^{reality} =$ search $testpkt$ in each $pkttrace_{port_i}$
        if $Ports_{testpkt}^{reality} == Ports_{testpkt}^{logical}$
            $context_i$ test pass
        else
            $context_i$ test fail
            ▷ Compare the $port$ of $Ports_{testpkt}^{reality}$ and $Ports_{testpkt}^{logical}$ and find the first different port, which is the NF that causes the failure.
            $FailedNFPort = FirstDiffPort(Ports_{testpkt}^{reality}, Ports_{testpkt}^{logical})$
        mark $context_i$ as tested
```

Figure 14: Pseudocode for BUZZ test resolution.

Minimizing Faulty Executions of Distributed Systems

Colin Scott[⋆] Aurojit Panda[⋆] Vjekoslav Brajkovic[⋄] George Necula[⋆]
Arvind Krishnamurthy[†] Scott Shenker[⋆⋄]
[⋆]*UC Berkeley* [⋄]*ICSI* [†]*University of Washington*

Abstract

When troubleshooting buggy executions of distributed systems, developers typically start by manually separating out events that are responsible for triggering the bug (signal) from those that are extraneous (noise). We present DEMi, a tool for automatically performing this minimization. We apply DEMi to buggy executions of two very different distributed systems, Raft and Spark, and find that it produces minimized executions that are between 1X and 4.6X the size of optimal executions.

1 Introduction

Even simple code can contain bugs (e.g., crashes due to unexpected input). But the developers of distributed systems face additional challenges, such as concurrency, asynchrony, and partial failure, which require them to consider all possible ways that non-determinism might manifest itself. Since the number of event orderings a distributed system may encounter grows exponentially with the number of events, bugs are commonplace.

Software developers discover bugs in several ways. Most commonly, they find them through unit and integration tests. These tests are ubiquitous, but they are limited to cases that developers anticipate themselves. To uncover unanticipated cases, semi-automated testing techniques such as fuzzing (where sequences of message deliveries, failures, etc. are injected into the system) are effective. Finally, despite pre-release testing, bugs may turn up once the code is deployed in production.

The last two means of bug discovery present a significant challenge to developers: the system can run for long periods before problems manifest themselves. The resulting executions can contain a large number of events, most of which are not relevant to triggering the bug. Understanding how a trace containing thousands of concurrent events lead the system to an unsafe state requires significant expertise, time,[1] and luck.

Faulty execution traces can be made easier to understand if they are first *minimized*, so that only events that are relevant to triggering the bug remain. In fact, developers often start troubleshooting by manually performing this minimization. Since developer time is typically much more costly than machine time, automated minimization tools for *sequential* test cases [24, 86, 94] have already proven themselves valuable, and are routinely applied to bug reports for software projects such as Firefox [1], LLVM [7], and GCC [6].

In this paper we address the problem of automatically minimizing executions of distributed systems. We focus on executions generated by fuzz testing, but we also illustrate how one might minimize production traces.

Distributed executions have two distinguishing features. Most importantly, input events (e.g., failures) are *interleaved* with internal events (e.g., intra-process message deliveries) of concurrent processes. Minimization algorithms must therefore consider both which input events and which (of the exponentially many) event schedules are likely to still trigger the bug. Our main contribution (discussed in section 3) is a set of techniques for searching through the space of event schedules in a timely manner; these techniques are inspired by our understanding of how practical systems behave.

Distributed systems also frequently exhibit non-determinism (e.g., since they make extensive use of timers to detect failures), complicating replay. We address this challenge (as we discuss in section 4) by instrumenting the Akka actor system framework [2] to gain nearly perfect control over when events occur.

With the exception of our prior work [70], we are unaware of any other tool that solves this problem without needing to analyze the code. Our prior work targeted a specific distributed system (SDN controllers), and focused on minimizing input events given limited control over the execution [70]. Here we target a broader range of systems, define the general problem of execution minimization, exercise significantly greater control, and systematically explore the state space. We also articulate new minimization strategies that quickly reduce input events, internal events, and message contents.

Our tool, Distributed Execution Minimizer (DEMi), is implemented in ~14,000 lines of Scala. We have applied DEMi to akka-raft [3], an open source Raft consensus implementation, and Apache Spark [90], a widely used data analytics framework. Across 10 known and discovered bugs, DEMi produces executions that are within a factor of 1X to 4.6X (1.6X median) the size of the smallest possible bug-triggering execution, and between

[1] Developers spend a significant portion of their time debugging (49% of their time according to one study [52]), especially when the bugs involve concurrency (70% of reported concurrency bugs in [37] took days to months to fix).

1X and 16X (4X median) smaller than the executions produced by the previous state-of-the-art blackbox technique [70]. The results we find for these two very different systems leave us optimistic that these techniques, along with adequate visibility into events (either through a framework like Akka, or through custom monitoring), can be applied successfully to a wider range of systems.

2 Problem Statement

We start by introducing a model of distributed systems as groundwork for defining our goals. As we discuss further in §4.2, we believe this model is general enough to capture the behavior of many practical systems.

2.1 System Model

Following [33], we model a distributed system as a collection of N single-threaded processes communicating through messages. Each process p has unbounded memory, and behaves deterministically according to a transition function of its current state and the messages it receives. The overall system S is defined by the transition function and initial configuration for each process.

Processes communicate by sending messages over a network. A message is a pair (p,m), where p is the identity of the destination process, and m is the message value. The network maintains a buffer of pending messages that have been sent but not yet delivered. Timers are modeled as messages a process can request to be delivered to itself at a specified later point in the execution.

A *configuration* of the system consists of the internal state of each process and the contents of the network's buffer. Initially the network buffer is empty.

An *event* moves the system from one configuration to another. Events can be one of two kinds. *Internal events* take place by removing a message m from the network's buffer and delivering it to the destination p. Then, depending on m and p's internal state, p enters a new internal state determined by its transition function, and sends a finite set of messages to other processes. Since processes are deterministic, internal transitions are completely determined by the contents of m and p's state.

Events can also be *external*. The three external events we consider are: process starts, which create a new process; forced restarts (crash-recoveries), which force a process to its initial state (though it may maintain nonvolatile state); and external message sends (p,m), which insert a message sent from outside the system into the network buffer (which may be delivered later as an internal event). We do not need to explicitly model fail-stop failures, since these are equivalent to permanently partitioning a process from all other processes.

A *schedule* is a finite sequence τ of events (both external and internal) that can be applied, in turn, starting from an initial configuration. Applying each event in the schedule results in an *execution*. We say that a schedule 'contains' a sequence of external events $E = [e_1, e_2, \ldots, e_n]$ if it includes only those external events (and no other external events) in the given order.

2.2 Testing

An *invariant* is a predicate P (a safety condition) over the internal state of all processes at a particular configuration C. We say that configuration C violates the invariant if $P(C)$ is false, denoted $\overline{P}(C)$.

A *test orchestrator* generates sequences of external events $E = [e_1, e_2, \ldots, e_n]$, executes them along with some (arbitrary) schedule of internal events, and checks whether any invariants were violated during the execution. The test orchestrator records the external events it injected, the violation it found, and the interleavings of internal events that appeared during the execution.

2.3 Problem Definition

We are given a schedule τ injected by a test orchestrator,[2] along with a specific invariant violation $\overline{P}$ observed at the end of the test orchestrator's execution.

Our main goal is to find a schedule containing a small sequence of external (input) events that reproduces the violation $\overline{P}$. Formally, we define a minimal causal sequence (MCS) to be a subsequence of external events $E' \sqsubseteq E$ such that there exists a schedule containing E' that produces $\overline{P}$, but if we were to remove any single external event e from E', there would not exist any schedules shorter[3] than τ containing $E' - e$ that produce $\overline{P}$.[4]

We start by minimizing external (input) events because they are the first level of abstraction that developers reason about. Occasionally, developers can understand the root cause simply by examining the external events.

For more difficult bugs, developers typically step through the internal events of the execution to understand more precisely how the system arrived at the unsafe state. To help with these cases, we turn to minimizing internal events after the external events have been minimized. At this stage we fix the external events and search for smaller schedules that still triggers the invariant violation, for example, by keeping some messages pending rather than delivering them. Lastly, we seek to minimize the contents (e.g. data payloads) of external messages.

Note that we do not focus on bugs involving only sequential computation (e.g. incorrect handling of unex-

[2] We explain how we obtain these schedules in §4.

[3] We limit the number of internal events to ensure that the search space is finite; any asynchronous distributed system that requires delivery acknowledgment is not guaranteed to stop sending messages [8], essentially because nodes cannot distinguish between crashes of their peers and indefinite message delays.

[4] It might be possible to reproduce $\overline{P}$ by removing multiple events from E', but checking all combinations is tantamount to enumerating its powerset. Following [94], we only require a 1-minimal subsequence E' instead of a globally minimal subsequence.

pected input), performance, or human misconfiguration. Those three bug types are more common than our focus: concurrency bugs. We target concurrency bugs because they are the most complex (correspondingly, they take considerably more time to debug [37]), and because mature debugging tools already exist for sequential code.

With a minimized execution in hand, the developer begins debugging. Echoing the benefits of sequential test case minimization, we claim that the greatly reduced size of the trace makes it easier to understand which code path contains the underlying bug, allowing the developer to focus on fixing the problematic code itself.

3 Approach

Conceptually, one could find MCSes by enumerating and executing every possible (valid, bounded) schedule containing the given external events. The globally minimal MCS would then be the shortest sequence containing the fewest external events that causes the safety violation. Unfortunately, the space of all schedules is exponentially large, so executing all possible schedules is not feasible. This leads us to our key challenge:

> *How can we maximize reduction of trace size within bounded time?*

To find MCSes in reasonable time, we split schedule exploration into two parts. We start by using delta debugging [94] (shown in Appendix A), a minimization algorithm similar to binary search, to prune extraneous external events. Delta debugging works by picking subsequences of external events, and checking whether it is possible to trigger the violation with just those external events starting from the initial configuration. We assume the user gives us a time budget, and we spread this budget evenly across each subsequence's exploration.

To check whether a particular subsequence of external events results in the safety violation, we need to explore the space of possible interleavings of internal events and external events. We use Dynamic Partial Order Reduction ('DPOR', shown in Appendix B) to prune this schedule space by eliminating equivalent schedules (i.e. schedules that differ only in the ordering of commutative events [34]). DPOR alone is insufficient though, since there are still exponentially many non-commutative schedules to explore. We therefore prioritize the order in which we explore the schedule space.

For any prioritization function we choose, an adversary could construct the program under test to behave in a way that prevents our prioritization from making any progress. In practice though, programmers do not construct adversarial programs, and test orchestrators do not construct adversarial inputs. We choose our prioritization order according to observations about how the programs we care about behave in practice.

Our central observation is that if one schedule triggers a violation, schedules that are similar in their causal structure should have a high probability of also triggering the violation. Translating this intuition into a prioritization function requires us to address our second challenge:

> *How can we reason about the similarity or dissimilarity of two different executions?*

We implement a hierarchy of *match* functions that tell us whether messages from the original execution correspond to the same logical message from the current execution. We start our exploration with a single, uniquely-defined schedule that closely resembles the original execution. If this schedule does not reproduce the violation, we begin exploring nearby schedules. We stop exploration once we have either successfully found a schedule resulting in the desired violation, or we have exhausted the time allocated for checking that subsequence.

External event minimization ends once the system has successfully explored all subsequences generated by delta debugging. Limiting schedule exploration to a fixed time budget allows minimization to finish in bounded time, albeit at the expense of completeness (i.e., we may not return a perfectly minimal event sequence).

To further minimize execution length, we continue to use the same schedule exploration procedure to minimize internal events once external event minimization has completed. Internal event minimization continues until no more events can be removed, or until the time budget for minimization as a whole is exhausted.

Thus, our strategy is to (i) pick subsequences with delta debugging, (ii) explore the execution of that subsequence with a modified version of DPOR, starting with a schedule that closely matches the original, and then by exploring nearby schedules, and (iii) once we have found a near-minimal MCS, we attempt to minimize the number of internal events. With this road map in mind, below we describe our minimization approach in greater detail.

3.1 Choosing Subsequences of External Events

We model the task of minimizing a sequence of external events E that causes an invariant violation as a function $ExtMin$ that repeatedly removes parts of E and invokes an oracle (defined in §3.2.1) to check whether the resulting subsequence, E', still triggers the violation. If E' triggers the violation, then we can assume that the parts of E removed to produce E' are not required for producing the violation and are thus not a part of the MCS.

$ExtMin$ can be trivially implemented by removing events one at a time from E, invoking the oracle at each iteration. However, this would require that we check $O(|E|)$ subsequences to determine whether each triggers the violation. Checking a subsequence is expensive, since it may require exploring a large set of event schedules. We therefore apply delta debugging [93, 94], an algorithm similar to binary search, to achieve $O(log(|E|))$ average case runtime (worst case $O(|E|)$). The delta debugging algorithm we use is shown in Appendix A.

Efficient implementations of *ExtMin* should not waste time trying to execute invalid (non-sensical) external event subsequences. We maintain validity by ensuring that forced restarts are always preceded by a start event for that process, and by assuming that external messages are independent of each other, i.e., we do not currently support external messages that, when removed, cause some other external event to become invalid. One could support minimization of dependent external messages by either requiring the user to provide a grammar, or by employing the $O(|E|^2)$ version of delta debugging that considers complements [94].

3.2 Checking External Event Subsequences

Whenever delta debugging selects an external event sequence E', we need to check whether E' can result in the invariant violation. This requires that we enumerate and check all schedules that contain E' as a subsequence. Since the number of possible schedules is exponential in the number of events, pruning this schedule space is essential to finishing in a timely manner.

As others have observed [38], many events occurring in a schedule are *commutative*, i.e., the system arrives at the same configuration regardless of the order events are applied. For example, consider two events e_1 and e_2, where e_1 is a message from process a being delivered to process c, and e_2 is a message from process b being delivered to process d. Assume that both e_1 and e_2 are co-enabled, meaning they are both pending at the same time and can be executed in either order. Since the events affect a disjoint set of nodes (e_1 changes the state at c, while e_2 changes the state at d), executing e_1 before e_2 causes the system to arrive at the same state it would arrive at if we had instead executed e_2 before e_1. e_1 and e_2 are therefore commutative. This example illustrates a form of commutativity captured by the happens-before relation [51]: two message delivery events a and b are commutative if they are concurrent, i.e. $a \not\rightarrow b$ and $b \not\rightarrow a$, and they affect a disjoint set of nodes.

Partial order reduction (POR) [34,38] is a well-studied technique for pruning commutative schedules from the search space. In the above example, given two schedules that only differ in the order in which e_1 and e_2 appear, POR would only explore one schedule. Dynamic POR (DPOR) [34] is a refinement of POR (shown in Appendix B): at each step, it picks a pending message to deliver, dynamically computes which other pending events are not concurrent with the message it just delivered, and sets backtrack points for each of these, which it will later use (when exploring other non-equivalent schedules) to try delivering the pending messages in place of the message that was just delivered.

Even when using DPOR, the task of enumerating all possible schedules containing E as a subsequence remains intractable. Moreover, others have found that naïve DPOR gets stuck exploring a small portion of the schedule space because of its depth-first exploration order [57]. We address this problem in two ways: first, as mentioned before, we limit *ExtMin* so it spreads its fixed time budget roughly evenly across checking whether each particular subsequence of external events reproduces the invariant violation. It does this by restricting DPOR to exploring a fixed number of schedules before giving up and declaring that an external event sequence does not produce the violation. Second, to maximize the probability that invariant violations are discovered quickly while exploring a fixed number of schedules, we employ a set of schedule exploration strategies to guide DPOR's exploration, which we describe next.

3.2.1 Schedule Exploration Strategies

We guide schedule exploration by manipulating two degrees of freedom within DPOR: (i) we prescribe which pending events DPOR initially executes, and (ii) we prioritize the order backtrack points are explored in. In its original form, DPOR only performs depth-first search starting from an arbitrary initial schedule, because it was designed to be *stateless* so that it can run indefinitely in order to find as many bugs as possible. Unlike the traditional use case, our goal is to minimize a known bug in a timely manner. By keeping some state tracking the schedules we have already explored, we can pick backtrack points in a prioritized (rather than depth-first) order without exploring redundant schedules.

A scheduling strategy implements a backtrack prioritization order. Scheduling strategies return the first violation-reproducing schedule they find (if any) within their time budget. We design our key strategy (shown in Algorithm 1) with the following observations in mind:

Observation #1: Stay close to the original execution. The original schedule provides us with a 'guide' for how we can lead the program down a code path that makes progress towards entering the same unsafe state. By choosing modified schedules that have causal structures that are close to the original schedule, we should have high probability of retriggering the violation.

We realize this observation by starting our exploration with a single, uniquely defined schedule for each external event subsequence: deliver only messages whose source, destination, and contents 'match' (described in detail below) those in the original execution, in the exact same order that they appeared in the original execution. If an internal message from the original execution is not pend-

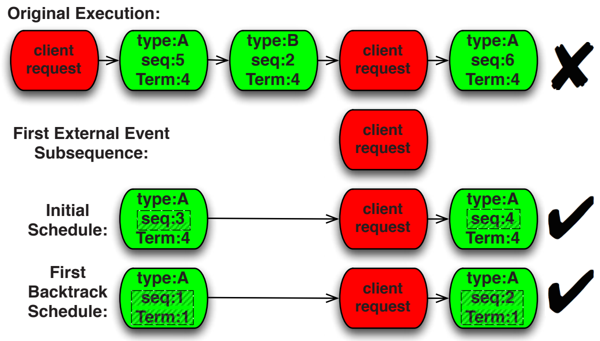


Figure 1: Example schedules. External message deliveries are shown in red, internal message deliveries in green. Pending messages, source addresses, and destination addresses are not shown. The 'B' message becomes absent when exploring the first subsequence of external events. We choose an initial schedule that is close to the original, except for the masked 'seq' field. The violation is not triggered after the initial schedule (depicted as ✔), so we next match messages by type, allowing us to deliver pending messages with smaller 'Term' numbers.

ing (i.e. sent previously by some actor) at the point that internal message should be delivered, we skip over it and move to the next message from the original execution. Similarly, we ignore any pending messages that do not match any events delivered in the original execution. In the case where multiple pending messages match, it does not matter which we choose (see Observation #2).

Matching Messages. A function *match* determines whether a pending message from a modified execution logically corresponds to a message delivered in the original execution. The simplest way to implement *match* is to check equality of the source, the destination, and all bytes of the message contents. Recall though that we are executing a *subsequence* of the original external events. In the modified execution the contents of many of the internal messages will likely change relative to message contents from the original execution. Consider, for example, sequence numbers that increment once for every message a process receives (shown as the 'seq' field in Figure 1). These differences in message contents prevent simple bitwise equality from finding many matches.

Observation #2: Data independence. Often, altered message contents such as differing sequence numbers do *not* affect the behavior of the program, at least with respect to whether the program will reach the unsafe state. Formally, this property is known as 'data-independence', meaning that the values of some message contents do not affect the system's control-flow [71, 82].

To leverage data independence, application developers can (optionally) supply us with a 'message fingerprint' function,[5] which given a message returns a string that depends on the relevant parts of the message, without considering fields that should be ignored when checking if two message instances from different executions refer to the same logical message. An example fingerprint function might ignore sequence numbers and authentication cookies, but concatenate the other fields of messages. Message fingerprints are useful both as a way of mitigating non-determinism, and as a way of reducing the number of schedules the scheduling strategy needs to explore (by drawing an equivalence relation between all schedules that only differ in their masked fields). We do not require strict data-independence in the formal sense [71]; the fields the user-defined fingerprint function masks over may in practice affect the control flow of the program, which is generally acceptable because we simply use this as a strategy to guide the choice of schedules, and can later fall back to exploring all schedules if we have enough remaining time budget.

We combine observations #1 and #2 to pick a single, unique schedule as the initial execution, defined by selecting pending events in the modified execution that *match* the original execution. This stage corresponds to the first two lines of TEST in Algorithm 1. We show an example initial schedule in Figure 1.

Challenge: history-dependent message contents. This initial schedule can be remarkably effective, as demonstrated by the fact that minimization often produces significant reduction even when we limit it to exploring this single schedule per external event subsequence. However, we find that without exploring additional schedules, the MCSes we find still contain extraneous events: when message contents depend on previous events, and the messages delivered in the original execution contained contents that depended on a large number of prior events, the initial schedule will remain inflated because it never includes "unexpected" pending messages that were not delivered in the original execution yet have contents that depend on fewer prior events.

To illustrate, let us consider two example faulty executions of the Raft consensus protocol. The first execution was problematic because all Raft messages contain logical clocks ("Term numbers") that indicate which epoch the messages belong to. The logical clocks are incremented every time there is a new leader election cycle. These logical clocks *cannot* be masked over by the message fingerprint, since they play an important role in determining the control flow of the program.

In the original faulty execution, the safety violation happened to occur at a point where logical clocks had high values, i.e. many leader election cycles had already taken place. We knew however that most of the leader election cycles in the beginning of the execution were not necessary to trigger the safety violation. Minimization restricted to only the initial schedule was *not* able to remove the earlier leader election cycles, though we

[5] It may be possible to extract message fingerprints automatically using program analysis or experimentation [77]. Nonetheless, manually defining fingerprints does not require much effort (see Table 4). Without a fingerprint function, we default to matching on message type (Observation #3).

Algorithm 1 Pseudocode for schedule exploration. TEST is invoked once per external event subsequence E'. We elide the details of DPOR for clarity (see Appendix B for a complete description). τ denotes the original schedule; b.counterpart denotes the message delivery that was delivered instead of b (variable m in the elif branch of STSSCHED); b.predecessors and b.successors denote the events occuring before and after b when b was set (τ''[0..i] and τ''[i+1...τ''.length] in STSSCHED).

```
backtracks ← {}
procedure TEST($E'$)
    STSSCHED($E'$,$\tau$)
    if execution reproduced ✘: return ✘
    while $\exists_{b \in backtracks.}$ b.type=b.counterpart.type ∧
            b.fingerprint ≠ b.counterpart.fingerprint ∧
            time budget for $E'$ not yet expired do
        reinitialize system, remove b from backtracks
        prefix ← b.predecessors + [ b ]
        if prefix (or superstring) already executed:
            continue
        STSSCHED($E'$,prefix + b.successors)
        if execution reproduced ✘: return ✘
    return ✔
procedure STSSCHED($E'$,$\tau'$)
    $\tau'' \leftarrow \tau'$.remove $\{e \mid e$ is external and $e \notin E'\}$
    for i from 0 to $\tau''$.length do
        if $\tau''$[i] is external:
            inject $\tau''$[i]
        elif $\exists_{m \in pending.}$ m.fingerprint = $\tau''$[i].fingerprint:
            deliver m, remove m from pending
            for m′ ∈ pending do
                if ¬commute(m,m′):
                    backtracks ← backtracks ∪ {m′}
```

would have been able to if we had instead delivered other pending messages with small term numbers.

The second execution was problematic because of *batching*. In Raft, the leader receives client commands, and after receiving each command, it replicates it to the other cluster members by sending them 'AppendEntries' messages. When the leader receives multiple client commands before it has successfully replicated them all, it batches them into a single AppendEntries message. Again, client commands cannot be masked over by the fingerprint function, and because AppendEntries are internal messages, we cannot shrink their contents.

We knew that the safety violation could be triggered with only one client command. Yet minimization restricted to only the initial schedule was unable to prune many client commands, because in the original faulty execution AppendEntries messages with large batch contents were delivered before pending AppendEntries messages with small batch contents.

These examples motivated our next observations:

Observation #3: Coarsen message matching. We would like to stay close to the original execution (per observation #1), yet the previous examples show that we should not restrict ourselves to schedules that only match according to the user-defined message fingerprints from the original execution. We can achieve both these goals by considering a more coarse-grained *match* function: the *type* of pending messages. By 'type', we mean the language-level type tag of the message object, which is available to the RPC layer at runtime.

We choose the next schedules to explore by looking for pending messages whose *types* (not contents) match those in the original execution, in the exact same order that they appeared in the original execution. We show an example in Figure 1, where any pending message of type 'A' with the same source and destination as the original messages would match. When searching for candidate schedules, if there are no pending messages that match the type of the message that was delivered at that step in the original execution, we skip to the next step. Similarly, we ignore any pending messages that do not match the corresponding type of the messages from the original execution. This leaves one remaining issue: how we handle cases where multiple pending messages match the corresponding original message's type.

Observation #4: Prioritize backtrack points that resolve match ambiguities. When there are multiple pending messages that match, we initially only pick one. DPOR (eventually) sets backtrack points for all other co-enabled dependent events (regardless of type or message contents). Of all these backtrack points, those that match the type of the corresponding message from the original trace should be most fruitful, because they keep the execution close to the causal structure of the original schedule except for small ambiguities in message contents.

We show the pseudocode implementing Observation #3 and Observation #4 as the while loop in Algorithm 1. Whenever we find a backtrack point (pending message) that matches the type but not the fingerprint of an original delivery event from τ, we replace the original delivery with the backtrack's pending message, and execute the events before and after the backtrack point as before.

Backtracking allow us to eventually explore all combinations of pending messages that match by type. Note here that we do not ignore the user-defined message fingerprint function: we only prioritize backtrack points for pending messages that have the same type *and* that differ in their message fingerprints.

Minimizing internal events. Once delta debugging over external events has completed, we attempt to further reduce the smallest reproducing schedule found so far. Here we apply delta debugging to internal events: for each subsequence of internal events chosen by delta debugging, we (i) mark those messages so that they are left pending and never delivered, and (ii) apply the same scheduling strategies described above for the remaining events to check whether the violation is still triggered.

Internal event minimization continues until there is no more minimization to be performed, or until the time budget for minimization as a whole is exhausted.

Observation #5: Shrink external message contents whenever possible. Our last observation is that the contents of external messages can affect execution length; because the test environment crafts these messages, it should minimize their contents whenever possible.

A prominent example is akka-raft's bootstrapping messages. akka-raft processes do not initially know which other processes are part of the cluster. They instead wait to receive an external bootstrapping message that informs them of the identities of all other processes. The contents of the bootstrapping messages (the processes in the cluster) determine *quorum size*: how many acknowledgments are needed to reach consensus, and hence how many messages need to be delivered. If the application developer provides us with a function for separating the components of such message contents, we can minimize their contents by iteratively removing elements, and checking to see if the violation is still triggerable until no single remaining element can be removed.

Recap. In summary, we first apply delta debugging ($ExtMin$) to prune external events. To check each external event subsequence chosen by delta debugging, we use a stateful version of DPOR. We first try exploring a uniquely defined schedule that closely matches the original execution. We leverage data independence by applying a user-defined message fingerprint function that masks over certain message contents. To overcome inflation due to history-dependent message contents, we explore subsequent schedules by choosing backtrack points according to a more coarse-grained match function: the types of messages. We spend the remaining time budget attempting to minimize internal events, and wherever possible, we seek to shrink external message contents.

3.3 Comparison to Prior Work

We made observations #1 and #2 in our prior work [70]. In this paper, we adapt observations #1 and #2 to determine the first schedule we explore for each external event subsequence (the first two lines of TEST). We refer to the scheduling strategy defined by these two observations as 'STSSched', named after the 'STS' system [70].

STSSched only prescribes a single schedule per external event subsequence chosen by delta debugging. In this work we systematically explore multiple schedules using the DPOR framework. We guide DPOR to explore schedules in a prioritized order based on similarity to the original execution (observations #3 and #4, shown as the while loop in TEST). We refer to the scheduling strategy used to prioritize subsequent schedules as 'TFB' (Type Fingerprints with Backtracks). We also minimize internal events, and shrink external message contents.

4 Systems Challenges

We implement our techniques in a publicly available tool we call DEMi (Distributed Execution Minimizer) [5]. DEMi is an extension to Akka [2], an actor framework for JVM-based languages. Actor frameworks closely match the system model in §2: actors are single-threaded entities that can only access local state and operate on messages received from the network one at a time. Upon receiving a message an actor performs computation, updates its local state and sends a finite set of messages to other actors before halting. Actors can be co-located on a single machine (though the actors are not aware of this fact) or distributed across multiple machines.

On a single machine Akka maintains a buffer of sent but not yet delivered messages, and a pool of message dispatch threads. Normally, Akka allows multiple actors to execute concurrently, and schedules message deliveries in a non-deterministic order. We use AspectJ [50], a mature interposition framework, to inject code into Akka that allows us to completely control when messages and timers are delivered to actors, thereby linearizing the sequence of events in an executing system. We currently run all actors on a single machine because this simplifies the design of DEMi, but minimization could also be distributed across multiple machines to improve scalability.

Our interposition lies above the network transport layer; DEMi makes delivery decisions for application-level (non-segmented) messages. If the application assumes ordering guarantees from the transport layer (e.g. TCP's FIFO delivery), DEMi adheres to these guarantees during testing and minimization to maintain soundness.

Fuzz testing with DEMi. We begin by using DEMi to generate faulty executions. Developers give DEMi a test configuration (we tabulate all programmer-provided specifications in Appendix C), which specifies an initial sequence of external events to inject before fuzzing, the types of external events to inject during fuzzing (along with probabilities to determine how often each event type is injected), the safety conditions to check (a user-defined predicate over the state of the actors), the scheduling constraints (e.g. TCP or UDP) DEMi should adhere to, the maximum execution steps to take, and optionally a message fingerprint function. If the application emits side-effects (e.g. by writing to disk), the test configuration specifies how to roll back side-effects (e.g. by deleting disk contents) at the end of each execution.

DEMi then repeatedly executes fuzz runs until it finds a safety violation. It starts by generating a sequence of random external events of the length specified by the configuration. DEMi then injects the initial set of external events specified by the developer, and then starts injecting external events from the random sequence. Developers can include special 'WaitCondition' markers in the initial set of events to execute, which cause DEMi to pause external event injection, and deliver pending internal messages at random until a specified condition holds, at which point the system resumes injecting external events. DEMi periodically checks invariants by halting the execution and invoking the developer-supplied safety predicate over the current state of all actors. Execution proceeds until a predicate violation is found, the supplied bound on execution steps is exceeded, or there are no more external or internal events to execute.

Once it finds a faulty execution DEMi saves a user-defined fingerprint of the violation it found (a violation fingerprint might, for example, mark which process(es) exhibited the violation),[6] a totally ordered recording of all events it executed, and information about which messages were sent in response to which events. Users can then replay the execution exactly, or instruct DEMi to minimize the execution as described in §3.

Mitigating non-determinism. Processes may behave non-deterministically. A process is non-deterministic if the messages it emits (modulo fingerprints) are not uniquely determined by the prefix of messages we have delivered to it in the past starting from its initial state.

The main way we control non-determinism is by interposing on Akka's API calls, which operate at a high level and cover most sources of non-determinism. For example, Akka provides a timer API that obviates the need for developers to read directly from the system clock.

Applications may also contain sources of non-determinism outside of the Akka API. We discovered the sources of non-determinism described below through trial and error: when replaying unmodified test executions, the violation was sometimes not reproduced. In these cases we compared discrepancies between executions until we isolated their source and interposed on it.

akka-raft instrumentation. Within akka-raft, actors use a pseudo random number generator to choose when to start leader elections. Here we provided a seeded random number generator under the control of DEMi.

Spark instrumentation. Within Spark, the task scheduler chooses the first value from a hashmap in order to decide what tasks to schedule. The values of the hashmap are arbitrarily ordered, and the order changes from execution to execution. We needed to modify Spark to sort the values of the hash map before choosing an element.

Spark runs threads ('TaskRunners') that are outside the control of Akka. These send status update messages to other actors during their execution. The key challenge with threads outside Akka's control is that we do not know when the thread has started and stopped each step of its computation; when replaying, we do not know how long to wait until the TaskRunner either resends an expected message, or we declare that message as absent.

We add two interposition points to TaskRunners: the start of the TaskRunner's execution, and the end of the TaskRunner's execution. At the start of the TaskRunner's execution, we signal to DEMi the identity of the TaskRunner, and DEMi records a 'start atomic block' event for that TaskRunner. During replay, DEMi blocks until the corresponding 'end atomic block' event to ensure that the TaskRunner has finished sending messages. This approach works because TaskRunners in Spark have a simple control flow, and TaskRunners do not communicate via shared memory. Were this not the case, we would have needed to interpose on the JVM's thread scheduler.

Besides TaskRunner threads, the Spark driver also runs a bootstrapping thread that starts up actors and sends initialization messages. We mark all messages sent during the initialization phase as 'unignorable', and we have DEMi wait indefinitely for these messages to be sent during replay before proceeding. When waiting for an 'unignorable' message, it is possible that the only pending messages in the network are repeating timers. We prevent DEMi from delivering infinite loops of timers while it awaits by detecting timer cycles, and not delivering more timers until it delivers a non-cycle message.

Spark names some of the files it writes to disk using a timestamp read from the system clock. We hardcode a timestamp in these cases to make replay deterministic.

Akka changes. In a few places within the Akka framework, Akka assigns IDs using an incrementing counter. This can be problematic during minimization, since the counter value may change as we remove events, and the (non-fingerprinted) message contents in the modified execution may change. We fix this by computing IDs based on a hash of the current callstack, along with task IDs in case of ambiguous callstack hashes. We found this mechanism to be sufficient for our case studies.

Stop-gap: replaying multiple times. In cases where it is difficult to locate the cause of non-determinism, good reduction can often still be achieved simply by configuring DEMi to replay each schedule multiple times and checking if any of the attempts triggered the safety violation.

Blocking operations. Akka deviates from the computational model we defined in §2 in one remaining aspect: Akka allows actors to block on certain operations. For example, actors may block until they receive a response to their most recently sent message. To deal with these cases we inject AspectJ interposition on blocking operations (which Akka has a special marker for), and signal to DEMi that the actor it just delivered a message to will not become unblocked until we deliver the response message. DEMi then chooses another actor to deliver a message to, and marks the previous actor as blocked until

[6] Violation fingerprints should be specific enough to disambiguate different bugs found during minimization, but they do not need to be specific to the exact state the system at the time of the violation. Less specific violation fingerprints are often better, since they allow DEMi to find divergent code paths that lead to the same buggy behavior.

DEMi decides to deliver the response.

4.1 Limitations

Safety vs. liveness. We are primarily focused on safety violations, not liveness or performance bugs.

Non-Atomic External Events. DEMi currently waits for external events (e.g. crash-recoveries) to complete before proceeding. This may prevent it from finding bugs involving finer-grained event interleavings.

Limited scale. DEMi is currently tied to a single physical machine, which limits the scale of systems it can test (but not the bugs it can uncover, since actors are unaware of colocation). We do not believe this is fundamental.

Shared memory & disk. In some systems processes communicate by writing to shared memory or disk rather than sending messages over the network. Although we do not currently support it, if we took the effort to add interposition to the runtime system (as in [74]) we could treat writes to shared memory or disk in the same way we treat messages. More generally, adapting the basic DPOR algorithm to shared memory systems has been well studied [34, 85], and we could adopt these approaches.

Non-determinism. Mitigating non-determinism in akka-raft and Spark required effort on our part. We might have adopted deterministic replay systems [29, 36, 56, 92] to avoid manual instrumentation. We did not because we could not find a suitably supported record and replay system that operates at the right level of abstraction for actor systems. Note, however that deterministic replay alone is not sufficient for minimization: deterministic replay does not inform how the schedule space should be explored; it only allows one to deterministically replay prefixes of events. Moreover, minimizing a single deterministic replay log (without exploring divergent schedules) yields executions that are orders of magnitude larger than those produced by DEMi, as we discuss in §6.

Support for production traces. DEMi does not currently support minimization of production executions. DEMi requires that execution recordings are complete (meaning all message deliveries and external events are recorded) and partially ordered. Our current implementation achieves these properties simply by testing and minimizing on a single physical machine.

To support recordings from production executions, it should be possible to capture partial orders without requiring logical clocks on all messages: because the actor model only allows actors to process a single message at a time, we can compute a partial order simply by reconstructing message lineage from per-actor event logs (which record the order of messages received and sent by each actor). Crash-stop failures do not need to be recorded, since from the perspective of other processes these are equivalent to network partitions. Crash-recovery failures would need to be recorded to disk. Byzantine failures are outside the scope of our work.

Recording a sufficiently detailed log for each actor adds some logging overhead, but this overhead could be modest. For the systems we examined, Akka is primarily used as a control-plane, *not* a data-plane (e.g. Spark does not send bulk data over Akka), where recording overhead is not especially problematic.

4.2 Generality

We distinguish between the generality of the DEMi artifact, and the generality of our scheduling strategies.

Generality of DEMi. We targeted the Akka actor framework for one reason: thanks to the actor API (and to a lesser extent, AspectJ), we did not need to exert much engineering effort to interpose on (i) communication between processes, (ii) blocking operations, (iii) clocks, and (iv) remaining sources of non-determinism.

We believe that with enough interposition, it should be possible to sufficiently control other systems, regardless of language or programming model. That said, the effort needed to interpose could certainly be significant.

One way to increase the generality of DEMi would be to interpose at a lower layer (e.g. the network or syscall layer) rather than the application layer. This has several limitations. First, some of our scheduling strategies depend on application semantics (e.g. message types) which would be difficult to access at a lower layer. Transport layer complexities would also increase the size of the schedule space. Lastly, some amount of application layer interposition would still be necessary, e.g. interposition on user-level threads or blocking operations.

Generality of scheduling strategies. At their core, distributed systems are just concurrent systems (with the additional complexities of partial failure and asynchrony). Regardless of whether they are designed for multi-core or a distributed setting, the key property we assume from the program under test is that small schedules that are similar to original schedule should be likely to trigger the same invariant violation. To be sure, one can always construct adversarial counterexamples. Yet our results for two very different types of systems leave us optimistic that these scheduling strategies are broadly applicable.

5 Evaluation

Our evaluation focuses on two key metrics: (i) the size of the reproducing sequence found by DEMi, and (ii) how quickly DEMi is able to make minimization progress within a fixed time budget. We show a high-level overview of our results in Table 1. The “Bug Type” column shows two pieces of information: whether the bug can be triggered using TCP semantics (denoted as “FIFO”) or whether it can only be triggered when UDP is used; and whether we discovered the bug ourselves or whether we reproduced a known bug. The “Provenance”

Bug Name	Bug Type	Initial	Provenance	STSSched	TFB	Optimal	NoDiverge
raft-45	Akka-FIFO, reproduced	2160 (E:108)	2138 (E:108)	1183 (E:8)	23 (E:8)	22 (E:8)	1826 (E:11)
raft-46	Akka-FIFO, reproduced	1250 (E:108)	1243 (E:108)	674 (E:8)	35 (E:8)	23 (E:6)	896 (E:9)
raft-56	Akka-FIFO, found	2380 (E:108)	2376 (E:108)	1427 (E:8)	82 (E:8)	21 (E:8)	2064 (E:9)
raft-58a	Akka-FIFO, found	2850 (E:108)	2824 (E:108)	953 (E:32)	226 (E:31)	51 (E:11)	2368 (E:35)
raft-58b	Akka-FIFO, found	1500 (E:208)	1496 (E:208)	164 (E:13)	40 (E:8)	28 (E:8)	1103 (E:13)
raft-42	Akka-FIFO, reproduced	1710 (E:208)	1695 (E:208)	1093 (E:39)	180 (E:21)	39 (E:16)	1264 (E:43)
raft-66	Akka-UDP, found	400 (E:68)	392 (E:68)	262 (E:23)	77 (E:15)	29 (E:10)	279 (E:23)
spark-2294	Akka-FIFO, reproduced	1000 (E:30)	886 (E:30)	43 (E:3)	40 (E:3)	25 (E:1)	43 (E:3)
spark-3150	Akka-FIFO, reproduced	600 (E:20)	536 (E:20)	18 (E:3)	14 (E:3)	11 (E:3)	18 (E:3)
spark-9256	Akka-FIFO, found (rare)	300 (E:20)	256 (E:20)	11 (E:1)	11 (E:1)	11 (E:1)	11 (E:1)

Table 1: Overview of case studies. "E:" is short for "Externals:". The 'Provenance', 'STSSched', and 'TFB' techniques are pipelined one after another. 'Initial' minus 'TFB' shows overall reduction; 'Provenance' shows how many events can be statically removed; 'STSSched' minus 'TFB' shows how our new techniques compare to the previous state of the art [70]; 'TFB' minus 'Optimal' shows how far from optimal our results are; and 'NoDiverge' shows the size of minimized executions when no divergent schedules are explored (explained in §6).

Bug Name	STSSched	TFB
raft-45	56s (594)	114s (2854)
raft-46	73s (384)	209s (4518)
raft-56	54s (524)	2078s (31149)
raft-58a	137s (624)	43345s (834972)
raft-58b	23s (340)	31s (1747)
raft-42	118s (568)	10558s (176517)
raft-66	14s (192)	334s (10334)
spark-2294	330s (248)	97s (78)
spark-3150	219s (174)	26s (21)
spark-9256	96s (73)	0s (0)

Table 2: Minimization runtime in seconds (total schedules executed). Overall runtime is the summation of "STSSched" and "TFB". spark-9256 only had unignorable events remaining after STSSched completed, so TFB was not necessary.

column shows how many events from the initial execution remain after statically pruning events that are concurrent with the safety violation. The "STSSched" column shows how many events remain after checking the initial schedules prescribed by our prior work [70] for each of delta debugging's subsequences. The "TFB" column shows the final execution size after we apply our techniques ('Type Fingerprints with Backtracks'), where we direct DPOR to explore as many backtrack points that match the types of original messages (but no other backtrack points) as possible within the 12 hour time budget we provided. Finally, the "Optimal" column shows the size of the smallest violation-producing execution we could construct by hand. We ran all experiments on a 2.8GHz Westmere processor with 16GB memory.

Overall we find that DEMi produces executions that are within a factor of 1X to 4.6X (1.6X median) the size of the smallest possible execution that triggers that bug, and between 1X and 16X (4X median) smaller than the executions produced by our previous technique (STSSched). STSSched is effective at minimizing external events (our primary minimization target) for most case studies. TFB is significantly more effective for minimizing internal events (our secondary target), especially for akka-raft. Replayable executions for all case studies are available at github.com/NetSys/demi-experiments.

We create the initial executions for all of our case studies by generating fuzz tests with DEMi (injecting a fixed number of random external events, and selecting internal messages to deliver in a random order) and selecting the first execution that triggers the invariant violation with ≥300 initial message deliveries. Fuzz testing terminated after finding a faulty execution within 10s of minutes for most of our case studies.

For case studies where the bug was previously known, we set up the initial test conditions (cluster configuration, external events) to closely match those described in the bug report. For cases where we discovered new bugs, we set up the test environment to explore situations that developers would likely encounter in production systems.

As noted in the introduction, the systems we focus on are akka-raft [3] and Apache Spark [90]. akka-raft, as an early-stage software project, demonstrates how DEMi can aid the development process. Our evaluation of Spark demonstrates that DEMi can be applied to complex, large scale distributed systems.

Reproducing Sequence Size. We compare the size of the minimized executions produced by DEMi against the smallest fault-inducing executions we could construct by hand (interactively instructing DEMi which messages to deliver). For 6 of our 10 case studies, DEMi was within a factor of 2 of optimal. There is still room for improvement however. For raft-58a for example, DEMi exhausted its time budget and produced an execution that was a factor of 4.6 from optimal. It could have found a smaller execution without exceeding its time budget with a better schedule exploration strategy.

Minimization Pace. To measure how quickly DEMi makes progress, we graph schedule size as a function of the number of executions DEMi tries. Figure 2 shows an example for raft-58b. The other case studies follow the same general pattern of sharply decreasing marginal gains.

We also show how much time (# of replays) DEMi took to reach completion of STSSched and TFB in Table 2.[7] The time budget we allotted to DEMi for all

[7] It is important to understand that DEMi is able to replay executions significantly more quickly than the original execution may have taken. This is because DEMi can trigger timer events before the wall-clock

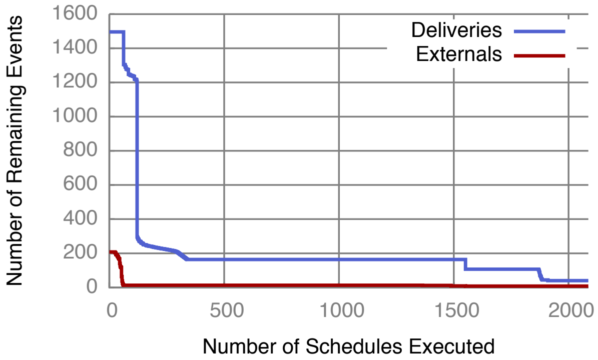


Figure 2: Minimization pace for raft-58b. Significant progress is made early on, then progress becomes rare.

case studies was 12 hours (43200s). All case studies except raft-56, raft-58a, and raft-42 reached completion of TFB in less than 10 minutes.

Qualitative Metrics. We do not evaluate how minimization helps with programmer productivity. Data on how humans do debugging is scarce; we are aware of only one study that measures how quickly developers debug minimized vs. non-minimized traces [40]. Nonetheless, since humans can only keep a small number of facts in working memory [62], minimization seems generally useful. As one developer puts it, "Automatically shrinking test cases to the minimal case is immensely helpful" [13].

5.1 Raft Case Studies

Our first set of case studies are taken from akka-raft [3]. akka-raft is implemented in 2,300 lines of Scala excluding tests. akka-raft has existing unit and integration tests, but it has not been deployed in production. The known bugs we reproduced had not yet been fixed; these were found by a recent manual audit of the code.

For full descriptions of each case study, see Appendix D. The lessons we took away from our akka-raft case studies are twofold. First, fuzz testing is quite effective for finding bugs in early-stage software. We found and fixed these bugs in less than two weeks, and several of the bugs would have been difficult to anticipate a priori. Second, debugging unminimized faulty executions would be very time consuming and conceptually challenging; we found that the most fruitful debugging process was to walk through events one-by-one to understand how the system arrived at the unsafe state, which would take hours for unminimized executions.

5.2 Spark Case Studies

Spark [4] is a mature software project, used widely in production. The version of Spark we used for our evaluation consists of more than 30,000 lines of Scala for just the core execution engine. Spark is also interesting because it has a significantly different communication pattern than Raft (e.g., statically defined masters).

	Without Shrinking	With shrinking
Initial Events	360 (E: 9 bootstraps)	360 (E: 9 bootstraps)
After STSSched	81 (E: 8 bootstraps)	51 (E: 5 bootstraps)

Table 3: External message shrinking results for raft-45 starting with 9 processes. Message shrinking + minimization was able to reduce the cluster size to 5 processes.

	akka-raft	Spark
Message Fingerprint	59	56
Non-Determinism	2	∼250
Invariants	331	151
Test Configuration	328	445

Table 4: Complexity (lines of Scala code) needed to define message fingerprints, mitigate non-determinism, define invariants, and configure DEMi. Akka API interposition (336 lines of AspectJ) is application independent.

For a description of our Spark case studies, see Appendix E. Our main takeaway from Spark is that for the simple Spark jobs we submitted, STSSched does surprisingly well. We believe this is because Spark's communication tasks were all almost entirely independent of each other. If we had submitted more complex Spark jobs with more dependencies between messages (e.g. jobs that make use of intermediate caching between stages) STSSched likely would not have performed as well.

5.3 Auxiliary Evaluation

External message shrinking. We demonstrate the benefits of external message shrinking with an akka-raft case study. Recall that akka-raft processes receive an external bootstrapping message that informs them of the IDs of all other processes. We started with a 9 node akka-raft cluster, where we triggered the raft-45 bug. We then shrank message contents by removing each element (process ID) of bootstrap messages, replaying these along with all other events in the failing execution, and checking whether the violation was still triggered. We were able to shrink the bootstrap message contents from 9 process IDs to 5 process IDs. Finally, we ran STSSched to completion, and compared the output to STSSched without the initial message shrinking. The results shown in Table 3 demonstrate that message shrinking can help minimize both external events and message contents.

Instrumentation Overhead. Table 4 shows the complexity in terms of lines of Scala code needed to define message fingerprint functions, mitigate non-determinism (with the application modifications described in §4), specify invariants, and configure DEMi. In total we spent roughly one person-month debugging non-determinism.

6 Related Work

We start this section with a discussion of the most closely related literature. We focus only on DEMi's minimization techniques, since DEMi's interposition and testing functionality is similar to other systems [55, 57, 72].

Input Minimization for Sequential Programs. Mini-

duration for those timers has actually passed, without the application being aware of this fact (cf. [39])

mization algorithms for sequentially processed inputs are well-studied [18,20,24,40,69,81,94]. These form a component of our solution, but they do not consider interleavings of internal events from concurrent processes.

Minimization without Interposition. Several tools minimize inputs to concurrent systems without controlling sources of non-determinism [10, 26, 44, 47, 76]. The most sophisticated of these replay each subsequence multiple times and check whether the violation is reproduced at least once [25,44]. Their major advantage is that they avoid the engineering effort required to interpose. However, we found in previous work [70] that bugs are often not easily reproducible without interposition.

QuickCheck's PULSE controls the message delivery schedule [25] and supports schedule minimization. During replay, it only considers the order messages are sent in, not message contents. When it cannot replay a step, it skips it (similar to STSSched), and reverts to random scheduling once expected messages are exhausted [43].

Thread Schedule Minimization. Other techniques seek to minimize thread interleavings leading up to concurrency bugs [22, 30, 41, 46]. These generally work by iteratively feeding a single input (the thread schedule) to a single entity (a deterministic scheduler). These approaches ensure that they never diverge from the original schedule (otherwise the recorded context switch points from the original execution would become useless). Besides minimizing context switches, these approaches at best *truncate* thread executions by having threads exit earlier than they did in the original execution.

Program Analysis. By analyzing the program's control- and dataflow dependencies, one can remove events in the middle of the deterministic replay log without causing divergence [19, 31, 42, 54, 74, 79]. These techniques do not explore alternate code paths. Program analysis also over-approximates reachability, disallowing them from removing dependencies that actually commute.

We compare against these techniques by configuring DEMi to minimize as before, but abort any execution where it detects a previously unobserved state transition. Column 'NoDiverge' of Table 1 shows the results, which demonstrate that divergent executions are crucial to DEMi's reduction gains for the akka-raft case studies.

Model Checking. Algorithmically, our work is most closely related to the model checking literature.

Abstract model checkers convert (concurrent) programs to logical formulas, find logical contradictions (invariant violations) using solvers, and minimize the logical conjunctions to aid understanding [23,49,61]. Model checkers are very powerful, but they are typically tied to a single language, and assume access to source code, whereas the systems we target (e.g. Spark) are composed of multiple languages and may use proprietary libraries.

It is also possible to extract formulas from raw binaries [11]. Fuzz testing is significantly lighter weight.

If, rather than randomly fuzzing, testers enumerated inputs of progressively larger sizes, failing tests would be minimal by construction. However, breadth first enumeration takes very long to get to 'interesting' inputs (After 24 hours of execution, our bounded DPOR implementation with depth bound slightly greater than the optimal trace size still had not found any invariant violations. In contrast, DEMi's randomized testing discovered most of our reported bugs within 10s of minutes). Furthermore, minimization is useful beyond testing, e.g. for simplifying production traces.

Because systematic input enumeration is intractable, many papers develop heuristics for finding bugs quickly [17, 28, 35, 55, 57, 63, 64, 66, 75, 78, 84]. We do the same, but crucially, we are able to use information from previously failing executions to guide our search.

As far as we know, we are the first to combine DPOR and delta debugging to minimize executions. Others have modified DPOR to keep state [87, 88] and to apply heuristics for choosing initial schedules [53], but these changes are intended to help find new bugs.

Bug Reproduction. Several papers seek to find a schedule that reproduces a given concurrency bug [9, 67, 91, 92]. These do not seek to find a minimal schedule.

Probabilistic Diagnosis. To avoid the runtime overhead of deterministic replay, other techniques capture carefully selected diagnostic information from production execution(s), and correlate this information to provide best guesses at the root causes of bugs [12,21,48,68,89]. We assume more complete runtime instrumentation (during testing), but provide exact reproducing scenarios.

Log Comprehension. Model inference techniques summarize log files in order to make them more easily understandable by humans [14–16,32,58,59]. Model inference is complementary, as it does not modify the event logs.

Program Slicing & Automated Debugging. Program slicing [80] and the subsequent literature on automated debugging [27, 45, 60, 73, 83, 95] seek to localize errors in the code itself. Our goal is to slice the temporal dimension of an execution rather than the code dimension.

7 Conclusion

Distributed systems, like most software systems, are becoming increasingly complex over time. In comparison to other areas of software engineering however, the development tools that help programmers cope with the complexity of distributed & concurrent systems are lagging behind their sequential counterparts. Inspired by the obvious utility of test case reduction tools, we sought to develop a minimization tool for distributed executions. Our evaluation results for two very different systems leave us optimistic that these techniques can be successfully applied to a wide range of concurrent systems.

References

[1] 7 Tips for Fuzzing Firefox More Effectively. https://blog.mozilla.org/security/2012/06/20/7-tips-for-fuzzing-firefox-more-effectively/.

[2] Akka official website. http://akka.io/.

[3] akka-raft Github repo. https://github.com/ktoso/akka-raft.

[4] Apache Spark Github repo. https://github.com/apache/spark/.

[5] DEMi Github repo. https://github.com/NetSys/demi.

[6] GNU's guide to testcase reduction. https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction.

[7] LLVM bugpoint tool: design and usage. http://llvm.org/docs/Bugpoint.html.

[8] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. International Workshop on Distributed Algorithms '97.

[9] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. SOSP '09.

[10] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq QuickCheck. Erlang '06.

[11] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. ICSE '14.

[12] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. OSDI '04.

[13] Basho Blog. QuickChecking Poolboy for Fun and Profit. http://tinyurl.com/qgc387k.

[14] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring Models of Concurrent Systems from Logs of their Behavior with CSight. ICSE '14.

[15] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. ESEC/FSE '11.

[16] A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of their Behavior. IEEE ToC '72.

[17] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. ASPLOS '10.

[18] M. Burger and A. Zeller. Minimizing Reproduction of Software Failures. ISSTA '11.

[19] Y. Cai and W. Chan. Lock Trace Reduction for Multithreaded Programs. TPDS '13.

[20] K.-h. Chang, V. Bertacco, and I. L. Markov. Simulation-Based Bug Trace Minimization with BMC-Based Refinement. IEEE TCAD '07.

[21] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, O. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. DSN '02.

[22] J. Choi and A. Zeller. Isolating Failure-Inducing Thread Schedules. SIGSOFT '02.

[23] J. Christ, E. Ermis, M. Schäf, and T. Wies. Flow-Sensitive Fault Localization. VMCAI '13.

[24] K. Claessen and J. Hughes. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. ICFP '00.

[25] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding Race Conditions in Erlang with QuickCheck and PULSE. ICFP '09.

[26] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. ICSE '07.

[27] H. Cleve and A. Zeller. Locating Causes of Program Failures. ICSE '05.

[28] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: Effective Unit Testing for Concurrency Libraries. PPoPP '10.

[29] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. OSDI '02.

[30] M. A. El-Zawawy and M. N. Alanazi. An Efficient Binary Technique for Frace Simplifications of Concurrent Programs. ICAST '14.

[31] A. Elyasov, I. W. B. Prasetya, and J. Hage. Guided Algebraic Specification Mining for Failure Simplification. TSS '13.

[32] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. IEEE ToSE '01.

[33] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. JACM '85.

[34] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. POPL '05.

[35] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. OSDI '14.

[36] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging For Distributed Applications. ATC '06.

[37] P. Godefroid and N. Nagappan. Concurrency at Microsoft - An Exploratory Survey. CAV '08.

[38] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* PhD Thesis, '95.

[39] D. Gupta, K. Yocum, M. Mcnett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. NSDI '06.

[40] M. Hammoudi, B. Burg, Gigon, and G. Rothermel. On the Use of Delta Debugging to Reduce Recordings and Facilitate Debugging of Web Applications. ESEC/FSE '15.

[41] J. Huang and C. Zhang. An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs. SAS '11.

[42] J. Huang and C. Zhang. LEAN: Simplifying Concurrency Bug Reproduction via Replay-Supported Execution Reduction. OOPSLA '12.

[43] J. M. Hughes. Personal Communication.

[44] J. M. Hughes and H. Bolinder. Testing a Database for Race Conditions with QuickCheck. Erlang '11.

[45] J. A. Jones and M. J. Harrold and J. Stasko. Visualization of Test Information To Assist Fault Localization. ICSE '02.

[46] N. Jalbert and K. Sen. A Trace Simplification Technique for Effective Debugging of Concurrent Programs. FSE '10.

[47] W. Jin and A. Orso. F3: Fault Localization for Field Failures. ISSTA '13.

[48] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures. SOSP '15.

[49] S. Khoshnood, M. Kusano, and C. Wang. ConcBugAssist: Constraint Solving for Diagnosis and Repair of Concurrency Bugs. ISSTA '15.

[50] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. ECOOP '01.

[51] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. CACM '78.

[52] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: a Study of Developer Work Habits. ICSE '06.

[53] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. FASE '10.

[54] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang. Toward Generating Reducible Replay Logs. PLDI '11.

[55] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. OSDI '14.

[56] C.-C. Lin, V. Jalaparti, M. Caesar, and J. Van der Merwe. DEFINED: Deterministic Execution for Interactive Control-Plane Debugging. ATC '13.

[57] H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. NSDI '09.

[58] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. ICSE '08.

[59] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining Invariants from Console Logs for System Problem Detection. ATC '10.

[60] M. Jose and R. Majmudar. Cause Clue Causes: Error Localization Using Maximum Satisfiability. PLDI '11.

[61] N. Machado, B. Lucia, and L. Rodrigues. Concurrency Debugging with Differential Schedule Projections. PLDI '15.

[62] G. A. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. Psychological Review '56.

[63] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. PLDI '07.

[64] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. SOSP '08.

[65] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. ATC '14.

[66] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from their Hiding Places. ASPLOS '09.

[67] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. SOSP '09.

[68] S. M. Park. *Effective Fault Localization Techniques for Concurrent Software*. PhD Thesis, '14.

[69] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case Reduction for C Compiler Bugs. PLDI '12.

[70] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. SIGCOMM '14.

[71] O. Shacham, E. Yahav, G. G. Gueta, A. Aiken, N. Bronson, M. Sagiv, and M. Vechev. Verifying Atomicity via Data Independence. ISSTA'14.

[72] J. Simsa, R. Bryant, and G. A. Gibson. dBug: Systematic Evaluation of Distributed Systems. SSV '10.

[73] W. Sumner and X. Zhang. Comparative Causality: Explaining the Differences Between Executions. ICSE '13.

[74] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling Tracing of Long-Running Multithreaded Programs via Dynamic Execution Reduction. ISSTA '07.

[75] V. Terragni, S.-C. Cheung, and C. Zhang. RECONTEST: Effective Regression Testing of Concurrent Programs. ICSE '15.

[76] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing Production Run Failures at the User's Site. SOSP '07.

[77] Twitter Blog. Diffy: Testing Services Without Writing Tests. https://blog.twitter.com/2015/diffy-testing-services-without-writing-tests.

[78] R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting Where it Hurts: An Automatic Concurrent Debugging Technique. ISSTA '07.

[79] J. Wang, W. Dou, C. Gao, and J. Wei. Fast Reproducing Web Application Errors. ISSRE '15.

[80] M. Weiser. Program Slicing. ICSE '81.

[81] A. Whitaker, R. Cox, and S. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. SOSP '04.

[82] P. Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic. POPL '86.

[83] J. Xuan and M. Monperrus. Test Case Purification for Improving Fault Localization. FSE '14.

[84] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. NSDI '09.

[85] M. Yabandeh and D. Kostic. DPOR-DS: Dynamic Partial Order Reduction in Distributed Systems. 2009 Tech Report.

[86] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. PLDI '11.

[87] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. MCS '08.

[88] X. Yi, J. Wang, and X. Yang. Stateful Dynamic Partial-Order Reduction. FMSE '06.

[89] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. ASPLOS '10.

[90] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI '12.

[91] C. Zamfir, G. Altekar, G. Candea, and I. Stoica. Debug Determinism: The Sweet Spot for Replay-Based Debugging. HotOS '11.

[92] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. EuroSys '10.

[93] A. Zeller. Yesterday, my program worked. Today, it does not. Why? ESEC/FSE '99.

[94] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE TSE '02.

[95] S. Zhang and C. Zhang. Software Bug Localization with Markov Logic. ICSE '14.

A Delta Debugging

We show the Delta Debugging simplification algorithm [93] we use in Figure 3, and an example execution of Delta Debugging in Figure 5. An updated version of the ddmin simplification algorithm appeared in [94]. We use the simpler version of ddmin (which is equivalent to the version ddmin from [94], except that it does not consider complements) because we ensure that each subsequence of external events is consistent (semantically valid), and therefore are still guarenteed to find a 1-minimal output without needing to consider complements.

B DPOR

We show the original depth-first version of Dynamic Partial Order Reduction in Algorithm 2. Our modified DPOR algorithm uses a priority queue rather than a (recursive) stack, and tracks which schedules it has explored in the past. Tracking which schedules we have explored in the past is necessary to avoid exploring redundant schedules (an artifact of our non depth-first exploration order). The memory footprint required for tracking previously explored schedules continues growing for every new schedule we explore. Because we assume a fixed time budget though, we typically exhaust our time budget before DEMi runs out of memory.

There are a few desirable properties of DPOR we want to maintain, despite our prioritized exploration order:

Soundness: any executed schedule should be valid, i.e. possible to execute on an uninstrumented version of the program starting from the initial configuration.

Step	External Event Subsequence								TEST
1	e_1	e_2	e_3	e_4	·	·	·	·	✔
2	·	·	·	·	e_5	e_6	e_7	e_8	✔
3	e_1	e_2	·	·	e_5	e_6	e_7	e_8	✔
4	·	·	e_3	e_4	e_5	e_6	e_7	e_8	✘
5	·	·	e_3	·	e_5	e_6	e_7	e_8	✘ (e_3 found)
6	e_1	e_2	e_3	e_4	e_5	e_6	·	·	✘
7	e_1	e_2	e_3	e_4	e_5	·	·	·	✔ (e_6 found)
Result	·	·	e_3	·	·	e_6	·	·	

Table 5: Example execution of Delta Debugging, taken from [93]. '·' denotes an excluded original external event.

Programmer-provided Specification	Default
Initial cluster configuration	-
External event probabilities	No external events
Invariants	Uncaught exceptions
Violation fingerprint	Match on any violation
Message fingerprint function	Match on message type
Non-determinism mitigation	Replay multiple times

Table 6: Tasks we assume the application programmer completes in order to test and minimize using DEMi. Defaults of '-' imply that the task is not optional.

Efficiency: the happens-before partial order for every executed schedule should never be a prefix of any other partial orders that have been previously explored.

Completeness: when the state space is acyclic, the strategy is guaranteed to find every possible safety violation.

Because we experimentally execute each schedule, soundness is easy to ensure (we simply ensure that we do not violate TCP semantics if the application assumes TCP, and we make sure that we cancel timers whenever the application asks to do so). Improved efficiency is the main contribution of partial order reduction. The last property—completeness—holds for our modified version of DPOR so long as we always set at least as many backtrack points as depth-first DPOR.

C Programmer Effort

In Table 6 we summarize the various tasks, both optional and necessary, that we assume programmers complete in order to test and minimize using DEMi.

D Raft Case Studies

Raft is a consensus protocol, designed to replicate a fault tolerant linearizable log of client operations. akka-raft is an open source implementation of Raft.

The external events we inject for akka-raft case studies are bootstrap messages (which processes use for discovery of cluster members) and client transaction requests. Crash-stop failures are indirectly triggered through fuzz schedules that emulate network partitions. The cluster size was 4 nodes (quorum size=3) for all akka-raft case studies.

The invariants we checked for akka-raft are the consensus invariants specified in Figure 3 of the Raft paper [65]: Election Safety (at most one leader can be elected in a given term), Log Matching (if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index), Leader Completeness (if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms), and State Machine Safety (if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index). Note that a violation of any of these invariants allows for the possibility for the system to later violate the main linearizability in-

Input: E s.t. E is a sequence of externals, and $\text{TEST}(E) = \text{✘}$. Output: $E' = ddmin(E)$ s.t. $E' \sqsubseteq E$, $\text{TEST}(E') = \text{✘}$, and E' is minimal.

$$ddmin(E) = ddmin_2(E, \emptyset) \quad \text{where}$$

$$ddmin_2(E', R) = \begin{cases} E' & \text{if } |E'| = 1 \text{ (“base case”)} \\ ddmin_2\big(E_1, R\big) & \text{else if TEST}(E_1 \cup R) = \text{✘ (“in } E_1\text{”)} \\ ddmin_2\big(E_2, R\big) & \text{else if TEST}(E_2 \cup R) = \text{✘ (“in } E_2\text{”)} \\ ddmin_2\big(E_1, E_2 \cup R\big) \cup ddmin_2\big(E_2, E_1 \cup R\big) & \text{otherwise (“interference”)} \end{cases}$$

where ✘ denotes an invariant violation, $E_1 \sqsubset E'$, $E_2 \sqsubset E'$, $E_1 \cup E_2 = E'$, $E_1 \cap E_2 = \emptyset$, and $|E_1| \approx |E_2| \approx |E'|/2$ hold.

Figure 3: Delta Debugging Algorithm from [93]. $\sqsubseteq$ and $\sqsubset$ denote subsequence relations. TEST is defined in Algorithm 1.

Algorithm 2 The original depth-first version of Dynamic Partial Order Reduction from [34]. $last(S)$ denotes the configuration reached after executing S; $next(\kappa, m)$ denotes the state transition (message delivery) where the message m is processed in configuration κ; $\rightarrow_S$ denotes ‘happens-before’; $pre(S,i)$ refers to the configuration where the transition t_i is executed; $dom(S)$ means the set $\{1,\ldots,n\}$; $S.t$ denotes S extended with an additional transition t.

```
Initially: EXPLORE($\emptyset$)
procedure EXPLORE($S$)
  $\kappa \leftarrow last(S)$
  for each message $m \in pending(\kappa)$ do
    if $\exists i = max(\{i \in dom(S) | S_i$ is dependent and may be coenabled with $next(\kappa, m)$ and $i \not\rightarrow_S m\})$:
      $E \leftarrow \{m' \in enabled(pre(S,i)) | m' = m$ or $\exists j \in dom(S) : j > i$ and $m' = msg(S_j)$ and $j \rightarrow_S m\}$
      if $E \neq \emptyset$:
        add any $m' \in E$ to $backtrack(pre(S,i))$
      else
        add all $m \in enabled(pre(S,i))$ to $backtrack(pre(S,i))$
  if $\exists m \in enabled(\kappa)$:
    $backtrack(\kappa) \leftarrow \{m\}$
    $done \leftarrow \emptyset$
    while $\exists m \in (backtrack(\kappa) \setminus done)$ do
      add $m$ to $done$
      EXPLORE($S.next(\kappa, m)$)
```

variant (State Machine Safety).

For each of the bugs where we did not initially know the root cause, we started debugging by first minimizing the failing execution. Then, we walked through the sequence of message deliveries in the minimized execution. At each step, we noted the current state of the actor receiving the message. Based on our knowledge of the way Raft is supposed to work, we found places in the execution that deviate from our understanding of correct behavior. We then examined the code to understand why it deviated, and came up with a fix. Finally, we replayed to verify the bug fix.

The akka-raft case studies in Table 1 are shown in the order that we found or reproduced them. To prevent bug causes from interfering with each other, we fixed all other known bugs for each case study. We reported all bugs and fixes to the akka-raft developers.

raft-45: Candidates accept duplicate votes from the same election term. Raft is specified as a state machine with three states: Follower, Candidate, and Leader. Candidates attempt to get themselves elected as leader by soliciting a quorum of votes from their peers in a given election term (epoch).

In one of our early fuzz runs, we found a violation of ‘Leader Safety’, i.e. two processes believed they were leader in the same election term. This is a highly problematic situation for Raft to be in, since the leaders may overwrite each others’ log entries, thereby violating the key linearizability guarantee that Raft is supposed to provide.

The root cause for this bug was that akka-raft’s candidate state did not detect duplicate votes from the same follower in the same election term. (A follower might resend votes because it believed that an earlier vote was dropped by the network). Upon receiving the duplicate vote, the candidate counts it as a new vote and steps up to leader before it actually achieved a quorum of votes.

raft-46: Processes neglect to ignore certain votes from previous terms. After fixing the previous bug, we found another execution where two leaders were elected in the same term.

In Raft, processes attach an ‘election term’ number to all messages they send. Receiving processes are supposed to ignore any messages that contain an election term that is lower than what they believe is the current term.

akka-raft properly ignored lagging term numbers for some, but not all message types. DEMi delayed the delivery of messages from previous terms and uncovered a case where a candidate incorrectly accepted a vote message from a previous election term.

raft-56: Nodes forget who they voted for. akka-raft is written as a finite state machine. When making a state transition, FSM processes specify both which state they want to transition to, and which instance variables they want to keep once they have transitioned.

All of the state transitions for akka-raft were correct except one: when the Candidate steps down to Follower (e.g., because it receives an 'AppendEntries' message, indicating that there is another leader in the cluster), it *forgets* which node it previously voted for in that term. Now, if another node requests a vote from it in the same term, it may vote for a different node than it previously voted for in the same term, later causing two leaders to be elected, i.e. a violation of Raft's "Leader Safety" condition. We discovered this by manually examining the state transitions made by each process throughout the minimized execution.

raft-58a: Pending client commands delivered before initialization occurs. After ironing out leader election issues, we started finding other issues. In one of our fuzz runs, we found that a leader process threw an assertion error.

When an akka-raft Candidate first makes the state transition to leader, it does not immediately initialize its state (the 'nextIndex' and 'matchIndex' variables). It instead sends a message to itself, and initializes its state when it receives that self-message.

Through fuzz testing, we found that it is possible that the Candidate could have pending ClientCommand messages in its mailbox, placed there before the Candidate transitioned to Leader and sent itself the initialization message. Once in the Leader state, the Akka runtime will first deliver the ClientCommand message. Upon processing the ClientCommand message the Leader tries to replicate it to the rest of the cluster, and updates its nextIndex hashmap. Next, when the Akka runtime delivers the initialization self-message, it will overwrite the value of nextIndex. When it reads from nextIndex later, it is possible for it to throw an assertion error because the nextIndex values are inconsistent with the contents of the Leader's log.

raft-58b: Ambiguous log indexing. In one of our fuzz tests, we found a case where the 'Log Matching' invariant was violated, i.e. log entries did not appear in the same order on all machines.

According to the Raft paper, followers should reject AppendEntries requests from leaders that are behind, i.e. prevLogIndex and prevLogTerm for the AppendEntries message are behind what the follower has in its log. The leader should continue decrementing its nextIndex hashmap until the followers stop rejecting its AppendEntries attempts.

This should have happened in akka-raft too, except for one hiccup: akka-raft decided to adopt 0-indexed logs, rather than 1-indexed logs as the paper suggests. This creates a problem: the initial value of prevLogIndex is ambiguous: Followers can not distinguish between an AppendEntries for an empty log (prevLogIndex == 0) an AppendEntries for the leader's 1st command (prevLogIndex == 0), and an AppendEntries for the leader's 2nd command (prevLogIndex == 1 1 == 0). The last two cases need to be distinguishable. Otherwise followers will not be able to reject inconsistent logs. This corner would have been hard to anticipate; at first glance it seems fine to adopt the convention that logs should be 0-indexed instead of 1-indexed.

As a result of this ambiguity, followers were unable to correctly reject AppendEntries requests from leader that were behind.

raft-42: Quorum computed incorrectly. We also found a fuzz test that ended in a violation of the 'Leader Completeness' invariant, i.e. a newly elected leader had a log that was irrecoverably inconsistent with the logs of previous leaders.

Leaders are supposed to commit log entries to their state machine when they knows that a quorum (N/2+1) of the processes in the cluster have that entry replicated in their logs. akka-raft had a bug where it computed the highest replicated log index incorrectly. First it sorted the values of matchIndex (which denote the highest log entry index known to be replicated on each peer). But rather than computing the median (or more specifically, the N/2+1'st) of the sorted entries, it computed the mode of the sorted entries. This caused the leader to commit entries too early, before a quorum actually had that entry replicated. In our fuzz test, message delays allowed another leader to become elected, but it did not have all committed entries in its log due to the previously leader committing too soon.

As we walked through the minimized execution, it became clear mid-way through the execution that not all entries were fully replicated when the master committed its first entry. Another process without all replicated entries then became leader, which constituted a violation of the "Leader Completeness" invariant.

raft-66: Followers unnecessarily overwrite log entries. The last issue we found is only possible to trigger if the underlying transport protocol is UDP, since it requires reorderings of messages between the same source, destination pair. The akka-raft developers say they do not currently support UDP, but they would like to adopt UDP in the future due to its lower latency.

The invariant violation here was a violation of the

'Leader Completeness' safety property, where a leader is elected that does not have all of the needed log entries.

Leaders replicate uncommitted ClientCommands to the rest of the cluster in batches. Suppose a follower with an empty log receives an AppendEntries containing two entries. The follower appends these to its log.

Then the follower subsequently receives an AppendEntries containing only the first of the previous two entries (this message was delayed). The follower will inadvertently delete the second entry from its log.

This is not just a performance issue: after receiving an ACK from the follower, the leader is under the impression that the follower has two entries in its log. The leader may have decided to commit both entries if a quorum was achieved. If another leader becomes elected, it will not necessarily have both committed entries in its log as it should, leading to a 'LeaderCompleteness' violation.

E Spark Case Studies

Spark is a large scale data analytics framework. We focused our efforts on reproducing known bugs in the core Spark engine, which is responsible for orchestrating computation across multiple machines.

We looked at the entire history of bugs reported for Spark's core engine. We found that most reported bugs only involve sequential computation on a single machine (e.g. crashes due to unexpected user input). We instead focused on reported bugs involving concurrency across machines or partial failures. Of the several dozen reported concurrency or partial failure bugs, we chose three.

The external events we inject for Spark case studies are worker join events (where worker nodes join the cluster and register themselves with the master), job submissions, and crash-recoveries of the master node. The Spark job we ran for all case studies was a simple parallel approximation of the digits of Pi.

spark-2294: Locality inversion. In Spark, an 'executor' is responsible for performing computation for Spark jobs. Spark jobs are assigned 'locality' preferences: the Spark scheduler is supposed to launch 'NODE_LOCAL' tasks (where the input data for the task is located on the same machine) before launching tasks without preferences. Tasks without locality preferences are in turn supposed to be launched before 'speculative' tasks.

The bug for this case study was the following: if an executor E is free, a task may be speculatively assigned to E when there are other tasks in the job that have not been launched (at all) yet. Similarly, a task without any locality preferences may be assigned to E when there was another 'NODE_LOCAL' task that could have been scheduled. The root cause of this bug was an error in Spark scheduler's logic: under certain configurations of pending Spark jobs and currently available executors, the Spark scheduler would incorrectly invert the locality priorities. We reproduced this bug by injecting random, concurrently running Spark jobs (with differing locality preferences) and random worker join events.

spark-3150: Simultaneous failure causes infinite restart loop. Spark's master node supports a 'Cold-Replication' mode, where it commits its state to a database (e.g., ZooKeeper). Whenever the master node crashes, the node that replaces it can read that information from the database to bootstrap its knowledge of the cluster state.

To trigger this bug, the master node and the driver process need to fail simultaneously. When the master node restarts, it tries to read its state from the database. When the driver crashes simultaneously, the information the master reads from the database is corrupted: some of the pointers referencing information about the driver are null. When the master reads this information, it dereferences a null pointer and crashes again. After failing, the master restarts, tries to recover its state, and crashes in an infinite cycle. The minimized execution for this bug contained exactly these 3 external events, which made the problematic code path immediately apparent.

spark-9256: Delayed message causes master crash. We found the following bug through fuzz testing.

As part of initialization, Spark's client driver registers with the Master node by repeatedly sending a RegisterApplication message until it receives a RegisteredApplication response. If the RegisteredApplication response is delayed by at least as long as the configured timeout value (or if the network duplicates the RegisterApplication RPC), it is possible for the Master to receive two RegisterApplication messages for the same client driver.

Upon receiving the second RegisterApplication message, the master attempts to persist information about the client driver to disk. Since the file containing information about the client driver already exists though, the master crashes with an IllegalStateException.

This bug is possible to trigger in production, but it will occur only very rarely. The name of the file containing information has a second-granularity timestamp associated with it, so it would only be possible to have a duplicate file if the second RegisteredApplication response arrived in the same second as the first response.

Acknowledgements

We thank our shepherd David Lie and the anonymous reviewers for their feedback. We also thank Peter Alvaro, Barath Raghavan, and Kay Ousterhout for feedback on the submitted draft. This research was supported by NSF CNS 1040838 and a gift from Intel. Colin Scott was also supported by an NSF Graduate Research Fellowship.

FlowRadar: A Better NetFlow for Data Centers

Yuliang Li* Rui Miao* Changhoon Kim† Minlan Yu*
*University of Southern California †Barefoot Networks

Abstract

NetFlow has been a widely used monitoring tool with a variety of applications. NetFlow maintains an active working set of flows in a hash table that supports flow insertion, collision resolution, and flow removing. This is hard to implement in merchant silicon at data center switches, which has limited per-packet processing time. Therefore, many NetFlow implementations and other monitoring solutions have to sample or select a subset of packets to monitor. In this paper, we observe the need to monitor all the flows without sampling in short time scales. Thus, we design FlowRadar, a new way to maintain flows and their counters that scales to a large number of flows with small memory and bandwidth overhead. The key idea of FlowRadar is to encode per-flow counters with a small memory and constant insertion time at switches, and then to leverage the computing power at the remote collector to perform network-wide decoding and analysis of the flow counters. Our evaluation shows that the memory usage of FlowRadar is close to traditional NetFlow with *perfect hashing*. With FlowRadar, operators can get better views into their networks as demonstrated by two new monitoring applications we build on top of FlowRadar.

1 Introduction

NetFlow [4] is a widely used monitoring tool for over 20 years, which records the flows (e.g., source IP, destination IP, source port, destination port, and protocol) and their properties (e.g., packet counters, and the flow starting and finish times). When a flow finishes after the inactive timeout, NetFlow exports the corresponding flow records to a remote collector. NetFlow has been used for a variety of monitoring applications such as accounting network usage, capacity planning, troubleshooting, and attack detection.

Despite its wide applications, the key problem to implement NetFlow in hardware is how to maintain an active working set of flows using a data structure with low time and space complexity. We need to handle collisions during flow insertion and remove old flows to make room for new ones. These tasks are challenging given the limited per-packet processing time at merchant silicon.

To handle this challenge, today's NetFlow is implemented in two ways: (1) Using complex custom silicon that is only available at high-end routers, which is too expensive for data centers; (2) Using software to count sampled packets from hardware, which takes too much CPU resources at switches. Because of the lack of usable NetFlow in data centers, operators have to mirror packets based on sampling or matching rules and analyze these packets in a remote collector [26, 40, 44, 34]. It is impossible to mirror all the packets because it takes too much bandwidth to mirror the traffic, and too many storage and computing resources at the remote collector to analyze every packet. (Section 2)

However, in data centers, there is an increasing need to have visibility of the counters for all the flows all the time. We need to cover all the flows to capture those transient loops, blackholes, and switch faults that only happen to a few flows in the Network and to perform fine-grained traffic analysis (e.g., anomaly detection). We need to cover these flows all the time to identify transient losses, bursts, and attacks in a timely fashion. (Section 3)

In this paper, we propose FlowRadar, which keeps counters for all the flows with low memory overhead and exports the flow counters in short time scales (e.g., 10 ms). The key design of FlowRadar is to identify the best division of labor between cheap switches with limited per-packet processing time and the remote collector with plenty of computing resources. We introduce *encoded flowsets* that only require simple constant-time instructions for each packet and thus are easy to implement with merchant silicon at cheap switches. We then decode these flowsets and perform network-wide analysis across time and switches all at the remote collector. We make the following key contributions in building FlowRadar:

Capture encoded flow counters with constant time for each packet at switches: We introduce encoded flowsets, which is an array of cells that encode the flows (5 tuples) and their counters. Encoded flowsets ensure constant per-packet processing time by *embracing* rather than handling hash collisions. It maps one flow to many cells, allows flows to collide in one cell, but ensure each cell has constant memory usage. Since encoded flowsets are small, we can afford to periodically export the entire flowsets to the remote collector in short time scales. Our encoded flowset data structure is an extension of Invertible Bloom filter Lookup Table (IBLT), but provides better support for counter updates.

Network-wide decoding and analysis at a remote collector: While each switch independently encodes the flows and counters, we observe that most flows traverse multiple switches. By leveraging the redundancies across switches, we make the encoded flowsets more compact. We then propose a network-wide decoding scheme to decode the flows and counters across switches. With the network-wide decoding, our encoded flowsets can reduce the amount of memory needed to track 100K flows by 5.6% compared to an ideal (and hence impractical) implementation of NetFlow with *perfect hashing* (i.e., no collisions) while providing 99% decoding success rate[1]. (Section 4 and 5)

FlowRadar can support a wide range of monitoring applications including both existing monitoring applications on NetFlow, and new ones that require monitoring all the flows all the time. As demonstrations, we design and build two systems on top of FlowRadar: one that detects transient loops and blackholes using a network-wide flow analysis and another that provides a per-flow loss map using temporal analysis (Section 6).

We discuss the implementation issues in Section 7, compare with related work in Section 8, and conclude in Section 9.

2 Motivation

In this section, we discuss the key challenges of implementing NetFlow. We then describe three alternative monitoring solutions (Table 1): NetFlow in high-end routers with custom silicon, NetFlow in cheap switches with merchant silicon, and selective mirroring. To address the limitations of these approaches, we present FlowRadar architecture, which identifies a good division of labor between the switches and the remote collector.

2.1 Key challenges of supporting NetFlow

Since NetFlow has been developed for over 20 years, there have been many implementations and extensions of NetFlow in routers and switches. We cannot capture all the NetFlow solutions here, and in fact many solutions are proprietary information. Instead, we focus on the basic function of NetFlow: storing the flow fields (e.g., 5 tuples) and the records (e.g., packet counter, flow starting time, the time that the flow is last seen, etc.) in a hash table. The key challenge is how to maintain the active working set of flows in the hash table given the limited packet processing time.

Maintain the active working set of flows: There are two key tasks in maintaining the active working set of flows:

(1) How to handle hash collisions during flow insertion? When we insert a new flow, it may experience collisions with existing flows. One solution is to store multiple flows in each cell in the hash table to reduce the chances of overflow (e.g., d-left hashing [14, 38]), which requires atomic many-byte memory accesses. Another solution to move existing flows around to make room for new flows (e.g., Cuckoo hashing [33]), which requires multiple, non-constant memory accesses per packet in the worst case. Both are very challenging to implement on merchant silicon with high line rate. The detailed challenges are discussed in Section 8.

(2) How to remove an old flow? We need to periodically remove old flows to make room for new flows in the hash table. If a TCP flow receives a FIN, we can remove it from the table. However, in data centers there are many persistent connections reused by multiple requests/responses or messages. To identify idle flows, NetFlow keeps the time a flow is last seen and periodically scan the entire hash table to check the inactive time of each flow. If a flow is inactive for more than the inactive timeout, NetFlow removes the flow and exports its counters. The inactive timeout can only be set between 10 and 600 seconds with a default value of 15 seconds [1]. When the hash table is large, it takes a significant time and switch CPU resources to scan the table and clean up the table entries.

Limited per-packet processing time at merchant silicon: It is hard to maintain the active working set of flows at the merchant silicon—the commodity switch design in data centers. The key constraint of the merchant silicon is the limited time we can spend on each packet. Suppose a switch has 40Gbps per port, which means 12ns per packet processing time for 64 Byte packets[2]. Let's assume the entire 12 ns can be dedicated to NetFlow by performing perfect packet pipelining and

[1] The decode success rate is defined as the probability of successfully decoding all the flows.

[2] This becomes worse when datacenters move to 100Gbps.

	Hardware-based NetFlow in custom silicon	**Sampled software-based NetFlow in merchant silicon**	**sFlow [40], EverFlow [44]**	**FlowRadar**
Division of labor				
state in switch hardware	active working set of flows	none	none	encoded flows and counters
state in switch software	none (or some active flows)	active working set of flows	none	none
data exported to collector	flow records after termination	flow records after termination	Selected pkts and timestamps	periodic encoded flow records
Coverage of traffic info				
Temporal coverage	No	No	No (if select control packets)	**Yes (milliseconds)**
Flow coverage	**All** or sampled packets	sampled packets	sampled or selected packets	**All**

Table 1: Comparing FlowRadar with hardware-based NetFlow in custom silicon, sampling-based software NetFlow in merchant silicon, and sFlow/EverFlow

allocating all other packet processing functions (packet header parsing, Layer 2/3 forwarding, ACLs, etc.) to other stages. Yet inside NetFlow, one needs to calculate the hash functions, look up SRAM, run a few ALU operations, and write back to the SRAM. Even with on-chip SRAM which has roughly 1 ns access time, to finish all these actions in 12 ns is still a challenge. (Similar arguments are made in [23] about the difficulties of implementing data streaming at routers.)

2.2 Alternative monitoring solutions

Due to the limited per-packet time in merchant silicon, one cannot process complex and non-constant time insertion and deletion actions as required in NetFlow. Therefore, there are three alternatives (Table 1):

Hardware-based NetFlow in custom silicon: One solution is to design custom silicon to maintain the active working set of flows in switch hardware. We can cache popular flow entries in on-chip SRAM, but the rest in off-chip SRAM or DRAM. We can also combine SRAM with expensive and power-hungry TCAM to support parallel lookup. Even with the expensive custom silicon, the test of Cisco high-end routers (Catalyst series) [18, 12] shows that there is still around 16% switch CPU overhead for storing 65K flow entries in hardware. Cisco highly recommends NetFlow users to choose sampling to reduce the NetFlow overhead on these routers [18].

Sampled software-based NetFlow in merchant silicon: Another solution is to sample packets and mirror them to the switch software, and maintain the active working set of flows in software. This solution works with cheap merchant silicon, but takes even more CPU overhead than hardware-based NetFlow in high-end routers. To reduce the switch CPU overhead of NetFlow and avoid interrupting other processes (e.g., OSPF, rule updates) in CPU, operators have to set sampling rate low enough (e.g., down to 1 in 4K). With such low sampling rate, operators cannot use NetFlow for fine-grained traffic analysis (e.g., anomaly detection) or capturing those events that only happen to some flows (e.g., transient loops or blackholes).

Selective mirroring (sFlow [40], EverFlow [44]): The final solution data center operators take today is to only sample packets or select packets based on match-action rules, and then mirror these packets to a remote collector. The remote collector extracts per flow information and performs detailed analysis. This solution works with existing merchant silicon, and best leverages the computing resources in the cloud. However, it takes too much bandwidth overhead to transfer all the packets to the collector and too much storage and computing overhead at the collector [44]. Therefore, operators can only get a partial view from the selected packets.

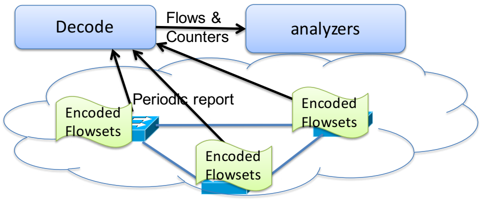


Figure 1: FlowRadar architecture

2.3 FlowRadar architecture

Instead of falling back to sampling in existing monitoring solutions, we aim at providing full visibility to all the flows all the time (see example use cases in Section 3). To achieve this, we propose to best leverage the capabilities at both the merchant silicon at switches and the computing power at the remote collector (Figure 1).

Capturing encoded flow counters at switches: FlowRadar chooses to encode flows and their counters into small fixed memory size that can be implemented in merchant silicon with constant flow insertion time. In this way, we can afford to capture all the flows without sampling, and periodically export these encoded flow counters to the remote collector in short time scales.

Decoding and analyzing flow counters at a remote collector: Given the encoded flows and counters exported from many switches, we can leverage the computing power at the remote collector to perform network-wide decoding of the flows, and temporal and flow space analysis for different monitoring applications.

3 Use cases

Since FlowRadar provides per flow counters, it can easily inherit many monitoring applications built on NetFlow such as accounting, capacity planning, application monitoring and profiling, and security analysis. In this section, we show that FlowRadar provides better monitoring support than sampled NetFlow and sFlow/EverFlow in two aspects: (1) Flow coverage: count all the flows without sampling; and (2) Temporal coverage: export these counters for each short time slot (e.g., 10 ms).

3.1 Flow coverage

Transient loop/blackhole detection: Transient loops and blackholes are important to detect, as they could cause packet loss. Just a few packet losses can cause significant tail-latency increase and throughput drops (especially because TCP treats losses as congestion signals) [31, 10], leading to violations of service level agreements (SLAs) and even a decrease of revenue [19, 39]. However, transient loops and blackholes are difficult to detect, as they may only affect a few packets during a very short time period. EverFlow or sampled NetFlow only select a few packets to monitor, and thus may miss most of the transient loops and blackholes. In addition, the transient loops and blackholes may only affect a certain kind of flows, so probing methods like Pingmesh [25] may not even notice the existence of them. Instead, if we can capture all the packets in each flow and maintain a corresponding counter in real time at every switch, we can quickly identify flows that are experiencing loops or blackholes (see Section 6).

Errors in match-action tables: Switches usually maintain a pipeline of match-action tables for packet processing. Data centers have reported table corruptions when switch memory experiences soft errors (i.e., bit flips) and these corruptions can lead to packet losses or incorrect forwarding for a small portion of the traffic [25, 44][3]. Such corruptions are hard to detect using network verification tools because they cannot see the actual corrupted tables. They are also hard to detect by sampled NetFlow or EverFlow because we cannot pre-decide the right set of packets to monitor. Instead, since FlowRadar can monitor all the packets, we can see problems when they happen (Section 6).

Fine-grained traffic analysis: Previous research has shown that packet sampling is inadequate for many fine-grained monitoring tasks such as understanding flow size distribution and anomaly detection [22, 20, 30]. Since FlowRadar monitors all the packets, we can provide more accurate traffic analysis and anomaly detection.

3.2 Temporal coverage

Per-flow loss map: Packet losses can be caused by a variety of reasons (e.g., congestion, switch interface bug, packet corruptions) and may have significant impact on applications. Although each TCP connection can detect its own losses (with sequence numbers or with switch support [17]), it is hard for the operators to understand where the losses happen inside the network, how many flows/applications are affected by such loss, and how the number of losses changes over time. NetFlow with low sampling rates cannot capture losses that happened to flows that are not sampled; and even for those sampled flows, we cannot infer losses from estimated flow counters. EverFlow can only capture control packets (e.g., NACK (Negative Acknowledgment)) to infer loss and congestion scenarios. Instead, if we can deploy FlowRadar at switches, we can directly get an overall map of the per-flow loss rate for all the flows soon after a burst of packets passes by (see Section 6).

Debugging ECMP load imbalance: ECMP load imbalance can lead to inefficient bandwidth usage in network and can significantly hurt application performance [11]. Short-term load imbalance can be caused by either (1) the network (e.g., ECMP not hashing on the right flow fields) or (2) the application (e.g., the application sends a sudden burst). If operators can quickly distinguish the two cases, they can make quick reactions to either reconfigure the ECMP functions for the network problem or to rate limit a specific application for the application problem.

EverFlow can diagnose some load imbalance problems by mirroring all the SYN and FIN packets and count the number of flows on each ECMP paths. However, it cannot diagnose either of the two cases above because it does not have detailed packet counters for each flow and does not know the traffic changes for these flows over time. Traditional NetFlow has similar limitations (i.e., no track of flows over time).

Timely attack detection: Some attacks exhibit specific temporal traffic patterns, which are hard to detect if we just count the number of packets per flow as NetFlow, or just capture the SYN/FIN packets as EverFlow. For example, TCP low-rate attacks [29] send a series of small traffic bursts that always trigger TCPs retransmission timeout, which can throttle TCP flows to a small fraction of the ideal rate. With per-flow counters at small time scale, we can not only detect these attacks by temporal analysis, but also report these attacks quickly (without waiting for the inactive timeout in NetFlow).

[3] For example, the L2 forwarding table gets corrupted. The packet that matches the entry can be flooded or mis-forwarded, leading to transient blackholes or loops before the entry is relearnt and corrected.

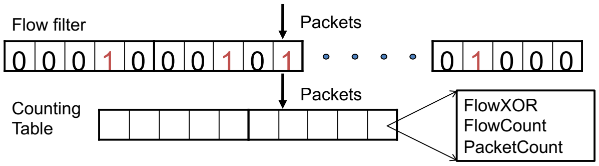


Figure 2: IBLT based flow counters

4 FlowRadar Design

The key design in FlowRadar is an encoding scheme to store flows and their counters in a small fixed-size memory, that requires constant insertion time at switches and can be decoded fast at the remote collector. When there is a sudden burst of flows, we can leverage network-wide decoding to decode more flows from multiple encoded flowsets. We also analyze the tradeoff between memory usage and decoding success rates.

4.1 Encoded Flowsets

The key challenge for NetFlow is how to handle flow collisions. Rather than designing solutions to *react* to flow collisions, our design focuses on how to *embrace* collisions: We allow flows to collide with each other without extra memory usage, and yet ensure we can decode individual flows and their counters at the collector.

There are two key designs that allow us to embrace collisions: (1) First, we hash the same flow to multiple locations (like Bloom filters). In this way, the chance that one flow collide with other flows in one of the bins decreases. (2) When multiple flows fall in the same cell, it is expensive to store them in a linked list. Instead, we use a XOR function to the packets of these flows without using extra bits. In this way, FlowRadar can work with a fixed-size memory space shared among many flows and has constant update and insertion time for all the flows.

Based on the two designs, the *encoded flowset* data structure is shown in Figure 2, which includes two parts: The first part is the *flow filter*. The flow filter is just a normal Bloom filter with an array of 0's and 1's, which is used for testing if a packet belongs to a new flow or not. The second part is the *counting table* which is used to store flow counters. The counting table includes the following fields:

- *FlowXOR:* which keeps the XOR of all the flows (defined based on 5 tuples) mapped in the bin
- *FlowCount:* which keeps the number of flows mapped in the bin
- *PacketCount:* which keeps the number of packets of all the flows mapped in the bin

Algorithm 1: FlowRadar packet processing

```
if $\exists\, i \in [1,k_f]$, s.t. $FlowFilter[H_i^F(p.flow)]$==0 then
    FlowFilter.add(p.flow);
    for $j$= $1..k_c$ do
        l = $H_j^C$(p.flow);
        CountTable[l].FlowXOR =
        CountTable[l].FlowXOR $\oplus$ p.flow;
        CountTable[l].FlowCount ++;
    end
end
for $j$= $1..k_c$ do
    CountTable[$H_j^C$(p.flow)].PacketCount ++;
end
```

As indicated in Algorithm 1, when a packet arrives, we first extract the flow fields of the packet, and check the flow filter to see if the flow has been stored in the flowset or not. If the packet comes from a new flow, we update the counting table by adding the packet's flow fields to FlowXOR and incrementing FlowCount and PacketCount at all the k_c locations. If the packet comes from an existing flow, we simply increment the packet counters at all the k_c locations.

Each switch sends the flowset to the collector every a few milliseconds, which we defined as *time slots*. In the rest of the paper, we set the value of the time slot to 10ms, unless explicitly setting it to other values in the context.

When FlowRadar collector receives the encoded flowset, it can decode the per flow counters by first looking for cells that include just one flow in it (called *pure cell*). For each flow in a *pure cell*, we perform the same hash functions to locate the other cells of this flow and remove it from all the cells (by XORing with the FlowXOR fields, subtracting the packet counter, and decrementing the flow counter). We then look for other *pure cells* and perform the same for the flows in each *pure cell*. The process ends when there are no *pure cells*. The detailed procedure is illustrated in Algorithm 3 in the appendix.

4.2 Network-wide decoding

Operators can configure the encoded flowset size based on the expected number of flows. However, there can be a sudden burst in terms of the number of flows. In that case, we may fail to decode some flows, when we do not have any cell with just one flow in the middle of the SingleDecode process. To handle a burst of flows, we propose a network-wide decoding scheme that can correlate multiple encoded flowsets at different switches to decode more flows. Our network-wide decoding process has two steps: decoding flows across switches and decoding counters inside a single switch.

FlowDecode across switches: The key observation is that if we use different hash functions at different switches, and if we cannot decode one flow in one encoded flowset, it is likely that we may be able to decode the flow at another encoded flowset at a different switch the flow traverses. For example, suppose we collect flowsets at two neighboring switches A_1 and A_2. We know that they have a common subset of flows from A_1 to A_2. Some of these flows may be single-decoded at A_1 but not A_2. If they match A_2's flow filter, we can remove these flows from A_2, which may lead to more one-flow cells. We can run SingleDecode on A_2 again.

Algorithm 2: FlowDecode

```
1  for $i$=1..N do
2      $S_i$ = SingleDecode($A_i$);
3  end
4  $finish$ = false;
5  while not $finish$ do
6      $finish$ = true;
7      foreach $A_i$,$A_j$ are neighbor do
8          foreach $flow$ in $S_i - S_j$ do
9              if $A_j$.FlowFilter.contains($flow$) then
10                 $S_j$.add($flow$);
11                 for $p$=1..$k_c$ do
12                     $l = H_p^{j,C}(flow)$;
13                     $A_j$.CountTable[$l$].FlowXOR =
                       $A_j$.CountTable[$l$].FlowXOR $\oplus$ flow;
14                     $A_j$.CountTable[$l$].FlowCount -= 1;
15                 end
16             end
17         end
18         foreach $flow$ in $S_j - S_i$ do
19             Update $S_i$ and $A_i$ same as $S_j$ and $A_j$
20         end
21     end
22     for $i$=1..N do
23         $result$ = SingleDecode($A_i$);
24         if $result \neq \emptyset$ then
25             $finish$ = false;
26         end
27         $S_i$.add($result$);
28     end
29 end
```

The general process of FlowDecode is described in Algorithm 2. Suppose we have the N encoded flowsets: $A_1..A_N$, and the corresponding sets of flows we get from SingleDecode $S_1..S_N$. For any two neighboring A_i and A_j, we check the all the flows we can decode from A_i but not A_j (i.e., $S_i - S_j$) to see if they also appear at A_j's flow filter. We remove those flows that match A_j's flow filter from A_j. We then run SingleDecode for all the flowsets again, get the new groups of $S_1..S_N$ and continue checking the neighboring pairs. We repeat the whole process until we cannot decode any more flows in the network.

Note that if we have the routing information of each packet, FlowDecode can speed up, because for one decoded flow at A_i, we only check the previous hop and next hop of A_i instead of all neighbors.

CounterDecode at a single switch: Although we can easily decode the flows using FlowDecode, we cannot decode the counters of them. This is because the counters at A and B for the same flow may not be the same due to the packet losses and on-the-fly packets (e.g. packets in A's output queue). Fortunately, from the FlowDecode process, we may already know all the flows in one encoded flowset. That is, at each cell, we know all the flows that are in the cell and the summary of these flows' counters. Formally, we know $CountTable[i].PacketCount = \sum_{\forall f, \exists j, H_j^C(f)=i} f.PacketCount$ for each cell i. Suppose the flowset has m_c cells and n flows, we have a total of m_c equations and n variables. This means we need to solve $MX = b$, where X is the vector of n variables and M and b are constructed from the above equations. We show how to construct M and b in Algorithm 4 in the Appendix.

Solving a large set of sparse linear equations is not easy. With the fastest solver lsqr (which is based on iteration) in Matlab, it takes more than 1 minute to get the counters for 100K flows. We speed up the computation from two aspects. First, we provide a close approximation of the counters, so that the solver can start from the approximation and reach the result fast. As the counters are very close across hops for the same flow, we can get the approximated counters during the FlowDecode. That is, when decoding A_i with the help of A_j's flows (Algorithm 2 line 7 to 21), we treat the counter from A_j as the counter in A_i for the same flow. We feed the approximated counters to the solver as initial values to start iteration, so that it can converge faster. Second, we use a loose stopping criterion for the iteration. As the counter is always an integer, we stop the iteration as long as the result is floating within a range of ± 0.5 around an integer. This significantly reduces the rounds of iteration. By these two optimizations, we reduce the computation time by around 70 times.

4.3 Analysis of decoding errors

SingleDecode: We now perform a formal analysis of the error rate in an encoded flowset. Suppose the flow filter uses k_f hash functions and m_f cells; and the counting table has k_c hash functions and m_c cells with s_c bits per cell. The total memory usage is $m_c \cdot s_c + m_f$. Assume there are n flows in the encoded flowset. For the flow filter, the false positive for a single new flow (i.e., the new flow being treated as an existing flow) is $(1 - e^{-k_f n/m_f})^{k_f}$. Thus the chance that none of the n flows experience false positives is $\prod_{i=1}^{n-1}(1 - (1 - e^{-k_f i/m_f})^{k_f})$. When the flow filter has a false positive, we can detect it by checking if there are non-zero PacketCounts after decoding. In this case the counters are not trustful, but we still get all the flows.

For the counting table, the decoding success rate of SingleDecode (i.e., the chance we can decode *all* the flows) is proved to be larger than $O(1-n^{-k_c+2})$, if $m_c > c_{k_c}n$, where c_{k_c} is a constant associates with k_c [24]. When we fail to decode some flows in the counting table, the already decoded flows and their counters are correct.

We choose to use separated flow filter and counting table rather than a combined one (i.e. the counting table also serves as a bloom filter to test new flow), because a combined one consumes much more memory. For a combined one, for each packet, we check the k_c cells it is hashed to, and view this flow as a new flow if and only if at least one of these k_c cells' FlowCount is 0. However, this solution requires far more memory than the separated solution. This is because for the counting table, a good parameter setting is about k_c = 3 and $m_c = 1.24n$ when n is larger than 10K based on the guidelines in [24] and our experiences in Section 5. In such a parameter setting, when we treat the counting table as a Bloom filter, the false positive rate for a new flow is $(1-e^{-k_c n/m_c})^{k_c}$ is larger than 99.9%. To keep the false positive rate low enough for all the n flows, we would have to significantly increase k_c and m_c.

NetDecode: We discuss FlowDecode and CounterDecode separately. For FlowDecode, we first consider a simple *pair-decode* case, where we run NetDecode between two nodes with the same set of flows. This can be viewed as decoding n flows in a large counting table with $2k_c$ hashes and $2m_c$ cells. This means we will need only half of the number of cells of the counting table with $2k_c$ hashes with SingleDecode. In our experiment, we only need $m_c = 8K$ for decoding 10K flow appear at both sides, which is even fewer than the number of flows.

For the more general network-wide FlowDecode, if all nodes in the network have more flows than expected and require FlowDecode, the decode success rate is similar to the *pair-decode* case. This is because for each node A, decoding its flows is similar to decoding the pair of A's flowset and the sum of flowsets from all the neighbors containing A's flows. However, it is more likely that only a portion of the nodes have more flows than expected, and the rest can SingleDecode. In this case, the decode success rate is higher than the *pair-decode* case.

For CounterDecode, we need at least the same number of linear equations as the number of variables (per flow counters). Because we have one equation per cell, we need the number of cells m_c to be at least the number of variables n. In practice, m_c should be slightly larger than the n, to obtain a high enough chance of having n linearly independent equations.

The complete NetDecode process is bottlenecked by CounterDecode not FlowDecode. This is because CounterDecode requires more memory and takes more time to decode. Since CounterDecode only runs on a single node, the memory usage and decoding speed of NetDecode at a node mostly depends on the number of flows in its own decoded flowset, rather than the number of other flowsets that contain similar flows.

5 Evaluation

In this section, we demonstrate that FlowRadar can scale to many flows and large networks with limited memory, bandwidth, and computing overhead, through simulations on FatTree topologies.

5.1 Scale to many flows

Parameter settings We set up a simulation network of FatTree with $k = 8$ (80 switches). We set the number of flows on each switch in 10 ms from 1K to 1000K. We generate an equal number of flows between each inter-Pod ToR pair. We then equally split these flows among ECMP paths. In this way, each switch has the same number of flows. We set the flow filter to ensure that the probability that one of the n flows experiences a false positive is 1/10 of the SingleDecode failure rate of the counting table. We set the optimal k_f and m_f according to the formulas in Section 4.3. We set $k_c = 4$ because it is the best for NetDecode. We select m_c based on the guidelines in [24]. We set the size of FlowCounter according to the expected number of flows. We conservatively set both NetFlow and FlowRadar packet counters as 4 Bytes, although in FlowRadar we collect statistics in a short time scale and thus would see much fewer packets and needs fewer bytes for the packet counter. Since our results are only related to the number of flows but not the packets, we generate a random set of flows as input.

We run decoding on 3.60GHz CPU cores, and parallelize decoding different flowsets on multiple cores.

The memory usage of FlowRadar is close to NetFlow with a perfect hash table: We first compare the memory usage between NetFlow and FlowRadar. As discussed in Section 2, it is almost impossible in merchant silicon to implement a hash-based design that handles flow insertions and collisions within the per packet time budget. If we implement a simple hash table, it would take 8.5TB to store 100K flows to ensure a 99% chance that there are no collisions. The actual data structure used in custom silicon would be proprietary information. Therefore, we compare with the best possible case for NetFlow—a perfect hash table without any collisions.

Even with a perfect hash table, NetFlow still needs to store in each cell the starting time of a flow and the time the flow is last seen for calculating inactive timeout (4 Bytes each). However, in FlowRadar, we do not need to keep timestamps in hardware because we use frequent

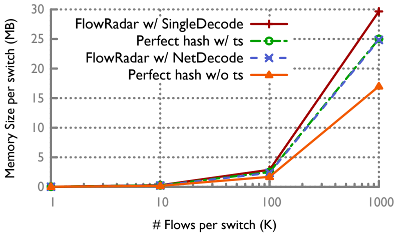


Figure 3: Memory usage per switch

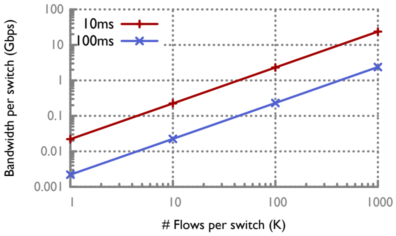


Figure 4: Bandwidth usage per switch

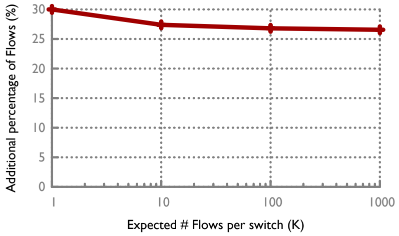


Figure 5: Extra #flows using NetDecode

reporting in a short scale. To fully decouple the benefit of FlowRadar data structure and removing timestamps, we also compare with perfect hashing without timestamps, which can be viewed as the optimal case we can reach.

Figure 3 shows that NetFlow with perfect hashing needs 2.5 MB per switch. FlowRadar needs only 2.88MB per switch with SingleDecode and 2.36MB per switch with NetDecode to store 100K flows with 99% decoding success[4], which is +15.2% and -5.6% compared to 2.5MB used by NetFlow. The best possible memory usage with perfect hashing without timestamps is 1.7MB per switch. With 1M flows, we need 29.7MB per switch for SingleDecode and 24.8MB per switch for NetDecode, which is +18.8% and -0.8% compared to NetFlow with perfect hashing and timestamps.

FlowRadar requires only a small portion of bandwidth to send encoded flowsets every 10ms. Figure 4 shows that we only need 2.3Gbps per switch to send encoded flowsets of 100K flows with 10ms time slot, and 0.23Gbps with 100ms time slot. In Facebook data center and traffic setting [35], a rack switch connects to 44 hosts with 10Gbps links, where each host send at most 100s to 1000s of concurrent flows in 5ms. Suppose there are a total of 2K*44 flows in 10ms in the rack switch, FlowRadar only incurs less than 0.52% of bandwidth overhead (2.3Gbps/(44*10Gbps)) with 10ms time slot.

FlowRadar with NetDecode can support 26.6-30% more flows than SingleDecode, with more decoding time Operators can configure FlowRadar based on the expected number of flows. When the number of flows goes beyond the expected number, we can use NetDecode to decode more flows given the same memory. Figure 5 shows with 1K to 1M expected number of flows, NetDecode can decode 26.6-30% more flows than SingleDecode given the same memory. So our solution can tolerate bursts in the number of flows.

Figure 6 shows the average decoding time of each flowset for the case with 100K expected flows. When the traffic is below 100K flows, the collector can run SingleDecode to quickly detect all the flows within 10 ms. When the traffic goes beyong 100K flows, we need NetDecode, which takes 283ms and 3275ms to decode flowsets with respective 101K flows and 126.8K flows.

We break down the NetDecode time into CounterDecode and FlowDecode. The result is shown in Figure 7. As the number of flows increases, the CounterDecode time increases fast, but the FlowDecode time remains low. If we just need to decode the flows, we need only 135ms, which is very small portion compared to CounterDecode's 3140ms. Note that the burst of flows does not always happen, so it is fine to wait for extra time to get the decoded flows and counters.

We do not rely on the routing information to reduce the NetDecode time, because it only helps reduce the FlowDecode time, which is only a small portion of the NetDecode time. The routing information can help reduce the FlowDecode time by 2 times.

5.2 Scale to many switches

We now investigate how FlowRadar scales with larger networks. For direct comparison, we assume the same number of flows per switch with different network sizes.

The memory and bandwidth usages per switch do not change with more switches: This is because the decoding success rate only relates to the number of flows and number of cells. Obviously this is true for SingleDecode. For NetDecode this is also true, because as long as all flows appear in at least 2 flowsets, NetDecode's decoding rate is similar no matter how many flowsets the flows appear in. The reason is that the bottleneck of the number of flows can be decoded is from CounterDecode, which is independent from other flowsets. For flowsets with 102.5K cells, two such flowsets can already decode more than 110K flows, but the CounterDecode can only support 100K flows (limited by the number of linearly independent equations).

Decoding requires proportionally more cores with more switches: The SingleDecode time per switch only relates to the number of flows in a flowset. For example, to decode 100K flows within 10ms, we need the same number of cores at the remote collector as the number of switches. This means for a network with 27K servers (K=48 FatTree) and 16 cores per server, we need about

[4] Note that even in the 1% of cases we cannot successfully decode all flows, we can still decode 61.7% of the flows on average.

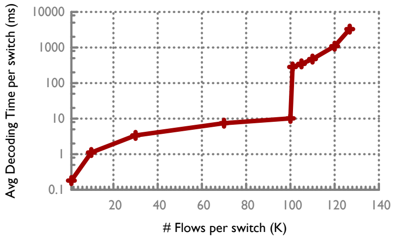


Figure 6: Decoding time

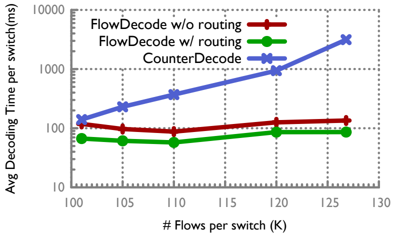


Figure 7: Breakdown of NetDecode Time

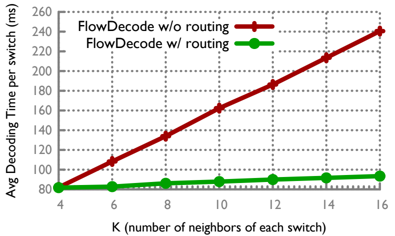


Figure 8: FlowDecode Time with different network size

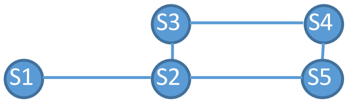


Figure 9: A flow path that has cycle

0.65% of the servers for the decoding.

NetDecode only happens during bursts of flows. The decoding time per switch increases slowly with more switches, because most time is spent on CounterDecode, which only relates to the number of flows in a flowset.

The FlowDecode time increases with larger networks, because it takes more time to check a decoded flow with the neighboring switches, when there are more neighbors in a larger network. In a FatTree network, suppose each switch has k neighbors. The total number of switches in the network is $n = \frac{5}{4}k^2$, so each flowset only checks with O($\sqrt{n}$) other flowsets. We tested the FlowDecode time with different FatTree network sizes by increasing k from 4 to 16. The memory on each switch is set expecting 100K flows for SingleDecode. We generate traffic such that the number of flows on each switch reaches the maximum number (126.8K) that could be NetDecoded. Figure 8 shows the result. The FlowDecode time increases linearly with k. However, it is still a small portion compared to CounterDecode time. For 126.8K flows per switch and $k = 16$ FatTree, FlowDecode only takes 0.24 seconds, which is 7.1% of the total decoding time. Routing information can speed up FlowDecode to 0.093 seconds, which is 2.9% of the total decoding time.

6 FlowRadar Analyzer

We show two use cases of FlowRadar: transient loop and blackhole detection with network-wide flow analysis and providing per-flow loss map with temporal analysis.

6.1 Transient loop/blackhole detection

With FlowRadar, we can infer the path for each flow by concatenating the switches that have records for that flow. As a result, we can easily provide a network-wide map of all the loops and blackholes, the time they happen, and the flows they affected.

Loops: We first identify all the switches that see the same flow during each time slot. If the switches form a cycle, then we suspect there is a loop. We cannot conclude that there is a loop because this may be caused by a routing change. For example, in Figure 9, we may observe counters at all the switches in one time slot with FlowRadar, which forms a cycle (S2,S3,S4,S5). However, this may be caused by a routing change from S1 → S2 → S5 to S1 → S2 → S3 → S4 → s5 within the time slot. To confirm, we need to compare the counter on the hop that is not in the cycle (counter1), and the counter on one hop in the cycle (counter2). If counter1 < counter2 then we can conclude that there is a loop. For example, if counter on S1 < counter on S3, we know this is a loop.

Blackholes: If a transient blackhole is longer than a slot's time, we can detect it by seeing the path of some flows stopped at some hop. If a transient blackhole is shorter than a slot's time, we still see a large difference between the counters before and after the blackhole at one slot. Note that we do not need the counters, but only the flow information to detect blackhole. Thus, during flow bursts, we can run FlowDecode without CounterDecode to detect blackholes faster.

Evaluation: We create a FatTree k=4 topology with 16 hosts and 20 switches in DeterLab [2]. We modify Open vSwitch [6] to support our traffic collection. We direct all the packets to the user space and maintain the encoded flowsets. We install forwarding rules for individual flows with different source and destination IP pair. We send persistent flows from each host to all the other hosts, which send one packet every 5 ms. This is to make sure that each flow has at least one packet in each time slot even if some packets is close to the slot's boundary.

We simulated a case that a VM migration causes a transient loop when the routing table on the edge switch S1 of the old VM location is updated so it sends packets up to the aggregation switch S2. But S2 has not been updated so it sends packets back to S1. We manually updated a rule at the edge switch S1 at around 10ms, which forms a loop S1 → S2 → S1, where S2 is an aggregation switch. We can detect the loop within 10ms.

To generate a blackhole, we manually remove a routing rule at an edge switch. We can detect the blackhole

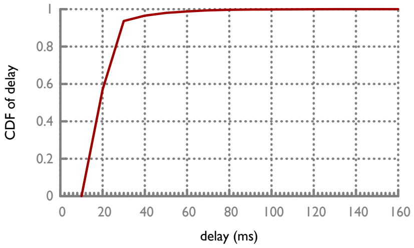


Figure 10: CDF of loss detection delay

within 20 ms. This is because there are still traffic in the first 10ms when the blackhole happens. So we can only confirm in the next 10ms.

6.2 Per-flow loss map

FlowRadar can generate a network-wide loss map by comparing the per-flow counters between the upstream and downstream switches (or hosts) in a sequence of time slots. A simple approach is that for each flow, the difference between the upstream and downstream counters is the number of losses in each time slot. However, this approach does not work in practice because it is impossible for the two switches capture exactly the same set of packets, even though today's data centers often have well synchronized clocks across switches at milliseconds level. This is because there are always packets on the fly between upstream and downstream switches (e.g., in the switch output queue).

To address this problem, we can wait until the flow finishes to compare its total number of packets at different hops. But this takes too long. Instead, we can detect losses faster by comparing counters for flowlets instead of flows. Suppose a time slot in FlowRadar is 10ms. We define *flowlets* as bursts of packets from a flow that are separated by gaps larger than a time slot [27]. With FlowRadar, we can identify flowlets between two time slots with counters equal to zero. Given a flowlet f, the upstream and downstream switches collect sequences of counters: $U_1...U_t$ and $D_1...D_t$ (D_0 and D_{t+1} are zero). We compute the total number of losses for the flowlet f as $\sum_{i=1}^{t}(U_i) - \sum_{i=1}^{t}(D_i)$. This is because if a packet does not arrive at the downstream switch for at least 10ms, it is very likely this packet is lost.

With this approach, we can get the accurate loss numbers and rates for all the flowlets that have finished. The key factor for our detection delay is the duration of flowlets. Fortunately, in data centers, many flows have short flowlets. For example, in a production web search workload [13], 87.5% of the partition/aggregate query flows are separated by a gap larger than 15 ms. 95% of query flows can finish within 10ms. Moreover, 95% of background large flows have 10-200 ms flow completion times with potential flowlets in them.

Evaluation: We evaluate our solution in a k=8 FatTree topology in a ns-3 simulator [5]. The FatTree has 128 hosts connected with 80 switches using 10G links. We take the same workload distribution from a production web search data center [13], but add the 1000 partition-aggregate queries per second with 20 incast degree (i.e., the number of responding nodes) and packet sizes of 1.5KB. The queue size of each port in our experiment is 150KB which means 100 packets of size 1.5KB. The flowlet durations are mostly shorter than 30ms with the maximum as 160ms. 50% of background traffic has 0ms interarrival time indicates application sends a spike of flows. The rest at least 40% of background traffic has interarrival time larger than 10ms for periodical update and short messages.

We run FlowRadar to collect encoded flowsets every 10ms at all the switches. We define detection delay as the time difference between when the loss happens and when we report it. Figure 10 shows the CDF of loss detection delay. We can detect more than 57% of the losses within 20ms, and more than 99% of the losses within 50ms.

7 Implementation

We now discuss the implementation issues in FlowRadar.

Encode and export flow counters at switches: FlowRadar only requires simple operations (e.g., hashing, XOR, and counting) that can be built on existing merchant silicon components. For example, hashing is already used in Layer 2 forwarding and ECMP functions. With the trend of programmable switches (e.g., P4 [8]), FlowRadar can be easier to implement.

We have implemented our prototype in P4 simulator [9], which will be released at [3]. We use an array of counters to store our counting table and flow filter. On each packet's arrival, we use the modify_field_with_hash_based_offset API to generate the k_c hash values for counting table and k_f hash values for flow filter, and use bit_xor API to xor the header into the flowXOR field. In the control plane, we use the stateful_read_counter API to read the content in our data.

Since the encoded flowset is small, we can export the entire encoded flowset to the collector rather than exporting them on a per flow basis. To avoid the interruptions on the data plane during the exporting phase, we can use two encoded flowset tables: the incoming packets update one table while we export data in another table. Note that there is a tradeoff between the memory usage and exporting overhead. If we export more often (with a smaller export interval), there are fewer flows in the interval and thus require fewer memory usage. Operators can configure the right export interval based on the number of flows in different time scales and the switch performance. For this paper, we set the time interval as 10 ms.

Deployment scenarios: Similar to NetFlow, we can deploy FlowRadar's encoded flowset either per port or per switch. The per-switch case would use less memory than per-port case because of multiplexing of flows. That is, it is unlikely that all the ports experience a burst in terms of the number of flows at the same time.

In the per-switch case, we still need to distinguish the incoming and outgoing flows (e.g., the two unidirectional flows in the same connection). One way to do this is to store the input port and output port as extra fields in the encoded flowset such as InputPortXOR and OutputPortXOR as what we did for the 5-tuple flow fields.[5] Another way is to maintain two encoded flowsets, one for incoming flows and another for outgoing flows.

FlowRadar can be deployed in any set of switches. FlowRadar can already report the per-flow counters in short time scales independently at each deployed switch. If FlowRadar is deployed at more switches, we can leverage network-wide decoding to handle more number of flows in a burst. Note that our network-wide decoding does not require full deployment. As long as there are flows that traverse two or more encoded flowsets, we start to gain benefits from network-wide decoding. Operators can choose where to deploy, and they know the flows where they deployed FlowRadar. In the ideal case, if all switches are deployed, then we know the per-flow counters at all locations, and the paths of the flows. Operators could also choose a subset of switches. For example, if we deploy only on ToR switches, the counters still cover all the events (e.g. loss) in the network, but we no longer know the exact locations where the flows appear in the network. As we mentioned in Section 5.2, the decoding success rate does not change as long as we have at least 2 flowsets, so partial deployment does not affect decoding success rate.

8 Related Work

8.1 Monitoring tools for data centers

Due to the problems of NetFlow, data center operators start to invent and use other monitoring tools. In addition to sFlow [40] and EverFlow [44], there are other in-network monitoring tools. OpenFlow [32] provide packet counters for each installed rules, which is only useful when the operators know which flows to track. Planck [34] leverages sampled mirroring at switches, which may not be sufficient for some monitoring tasks we discussed in Section 2. There are also many end-host based monitoring solutions such as SNAP which captures TCP-level statistics [41] and pingmesh [25] which leverages active probes. FlowRadar is complementary to the end-host based solutions by providing in-network view for individual flows.

8.2 Measurement data structures

There have been many hash-based data structures for measurement. Compared to them, FlowRadar has three unique features: (1) Store flow-counter pairs for many flows; (2) Easy to implement in merchant silicon; (3) Support network-wide decoding across switches.

Data structures for performance measurement and volume counting: Varghese et. al. proposed a group of data structures for loss, latency, and burst measurement [28, 37]. However, none of these solutions can maintain per flow metrics and scale to a large number of flows. There are many hash-based data structures that can keep per-flow state with small memory [15, 42, 36, 43]. However, most of them do not suit for NetFlow because they can only keep the values (i.e., per flow state). Instead, FlowRadar provides the key-value pairs (i.e., the flow tuples and the packet counters) and can scale to a large number of flows.

Hash-based data structures for storing key-value pairs: Cuckoo hashing [33] and d-left hashing [14, 38] are two hash table designs that can store key-value pairs with low memory usage. However, both are hard to implement in merchant silicon for NetFlow. This is because NetFlow requires inserting a flow immediately for an incoming packet so that follow up packets can update the same entry (i.e., *atomic read-update* operations). Otherwise, if one packet reads a cell that is being updated by a preceding packet, the counters become incorrect. Today, merchant silicon already has transactional memory that supports *read-update* operations in an atomic way for counters. However, typical merchant silicon can handle read-update operations against only a few (up to four) 4B- or 8B-long counters for each packet[6]. This is because to support high link rate of merchant silicon (typically a few Tbps today), merchant silicon must resort to a highly-parallelized packet-processing design, and the atomic-execution logic is at odds with such parallelism. In fact, to support such atomic read-update semantics for a small number of counters, merchant silicon has to employ various complicated hardware logic similar to operand forward [7].

A d-way Cuckoo hash table [33] hashes each key to d positions and stores the key in one of the empty positions. When all the d positions are full, we need to rebuild the table by moving items around to make room for the new key. However, this rebuilding process can only be implemented with switch software (i.e., the control plane),

[5] Similarly, one can easily add other flow properties (e.g., VLAN) as XOR sum fields.

[6] Note the total number of counters can still be larger; only the number of concurrently read-and-updatable counters is small.

because it requires multiple, and often-unbounded number of memory accesses [33]. Running the rebuilding process in switch software is not suitable for NetFlow, because NetFlow requires atomic read-update semantics.

d-left hashing splits a hash table with n buckets into d equal subtables each with n/d buckets, where each bucket contains L cells to hold up to L keys. d-left hashes a new key to d buckets, one in each subtable, and put the key in the bucket with the least load, breaking ties to the left. d-left requires first reading all Ld cells and testing if there is any match for an incoming flow. If there is a match, we increment the counter; otherwise, we put a new entry in an empty cell in the least-loaded bucket. There are two key challenges in supporting d-left: First, rather than read-update operations, d-left requires *atomic read-test-update* operations. The testing logic requires not only more ALUs and MUXes but also significantly increase the complexity of the atomic operation logic, making the critical section much longer in time. Second, d-left can only make insertion decisions after the testing on all Ld cells (each cell with 13 bytes 5-tuple fields and 4 bytes counter) are finished, which also increases the size of the atomic operation logic. Longer atomic operation duration can be a disaster for highly parallelized packet processing in merchant silicon.

In contrast, FlowRadar is easier to implement in merchant silicon, because of three reasons: First, FlowRadar only requires *atomic read-update* operations (i.e., increment/xor) rather than *atomic read-test-update*, which is much simpler in silicon design and has shorter atomic operation duration. Second, FlowRadar only requires atomic operations on a single cell and packets can update different cells in parallel. Thus FlowRadar requires significantly shorter atomic operations and is better fit for merchant silicon with high line rate.

It is impossible to support d-left with today's merchant silicon because the smallest d-left configuration (i.e., $d = 4$ and $L = 1$) needs to atomically read-test-update 4*17=68B, but today's silicon only supports 4*8B=32B. Thus, we compare FlowRadar with the basic d-left setting (i.e., $d = 4$ and $L = 1$) that may be supported in future silicon, and the setting recommended by [16] (i.e., $d = 3$ and $L = 5$) which is even harder to implement. To hold 100K flows on a memory of 2.74MB, the basic d-left has an overflow rate of 1.04%; both FlowRadar and the recommended d-left have no overflow. During flow bursts, FlowRadar can still report flows even when the counters cannot be decoded. Such flow information can be used for a variety of tasks like transient blackhole detection, route verification, and flow duration measurement. For example, to hold 152K flows in 2.74MB memory, the basic d-left has an overflow rate of 10%; the recommeded d-left has an overflow rate of 1.2%; FlowRadar can still decode all 152K flows (but not their counters).

Invertible Bloom filter Lookup Table (IBLT): FlowRadar is inspired by Invertible Bloom filter (IBF) [21] and Invertible Bloom filter Lookup Table (IBLT) [24]. IBF is used to keep a set of items. By comparing two IBFs, one can easily extract the differences between two sets. Rather then keeping a set of elements, FlowRadar needs to collect a key-value store of flows and their packet counters.

IBLT is an extension of IBF that can store key-value stores. Our counting table is built upon IBLT, but has two key extensions: *(1) How to handle value updates.* Since IBLT does not have a flow filter before it to identify if a key is new or old, it treats an existing key with a new value as a new key-value pair which has duplicated keys with existing key-value pairs. It then uses an arithmetic sum instead of a XOR sum in FlowXOR field, and a sum of hash values of the flows instead of a simple flow counter. This design takes more bits in both FlowXOR and FlowCount fields, which takes as much memory as FlowRadar uses for the flow filter. It also requires computations over large numbers (beyond 64bit integer), and more complex hash functions. Our experiments show that IBLT saves only 2.6% of memory for 100K keys but at the expense of 4.6 times more decoding time. *(2) How to decode the keys.* Our single node encoding scheme is similar to IBLT's, but takes much less time because of the simple FlowXOR and FlowCount fields. Moreover, with an extra flow filter, we support network-wide flow and counter decoding across multiple encoded flowsets.

9 Conclusion

We present FlowRadar, a new way to provide per-flow counters for all the flows in short time scales, which provides better visibility in data center networks. FlowRadar encodes flows and their counters with a small memory and constant insertion time at switches. It then introduces network-wide decoding of flowsets across switches to handle bursts of flows with limited memory. Our design can be improved in many aspects to further reduce the cost of computation, memory, and bandwidth, such as reducing the NetDecode time and better ways to leveraging redundancies across switch hops.

10 Acknowledgment

We thank our shepherd Sujata Banerjee, George Varghese, and the anonymous reviewers for their helpful feedbacks. This paper is partially supported by CNS-1453662, CNS-1423505, CNS-1413972, NSFC-61432009, and Google.

APPENDIX

A Algorithms

Algorithm 3: Decoding at a single node

```
Function SingleDecode(A)
    flowset = ∅;
    foreach c where CountTable[c].FlowCount==1 do
        flow = A.CountTable[c].FlowXOR;
        flowset.add(flow);
        count = A.CountTable[c].PacketCount;
        for j=1..$k_c$ do
            l=$H_j^C$(flow);
            A.CountTable[l].FlowXOR =
            CountTable[l].FlowXOR $\oplus$ flow;
            A.CountTable[l].FlowCount -= 1;
            A.CountTable[l].PacketCount -= count;
        end
    end
    return flowset;
```

Algorithm 4: Linear equations for CounterDecode

```
Function ConstructLinearEquations(A,S)
    M=ZeroMatrix; b=ColumnVector;
    foreach $flow_t$ in S do
        for j=1..$k_c$ do
            $l = H_j^C(flow_t)$; M[$l$,$t$] = 1;
        end
    end
    foreach CountTable[j] in A do
        b[$j$] = CountTable[$j$].PacketCount;
    end
```

References

[1] http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html.

[2] deterlab.net.

[3] Flowradar implementation in p4. https://github.com/USC-NSL/FlowRadar-P4.

[4] NetFlow. https://www.ietf.org/rfc/rfc3954.txt.

[5] ns-3 simulator. https://www.nsnam.org/.

[6] Open vSwitch. http://openvswitch.org/.

[7] Operand forwarding. https://en.wikipedia.org/wiki/Operand_forwarding.

[8] P4 language consortium. p4.org.

[9] P4 simulator. https://github.com/p4lang.

[10] Packet loss impact on tcp throughput in esnet. http://fasterdata.es.net/network-tuning/tcp-issues-explained/packet-loss/.

[11] Solving the mystery of link imbalance a metastable failure state at scale. https://code.facebook.com/posts/1499322996995183/.

[12] Router overhead when enabling netflow. http://blog.tmcnet.com/advanced-netflow-traffic-analysis/2013/05/router-overhead-when-enabling-netflow.html, 2013.

[13] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[14] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1), 1999.

[15] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond bloom filters: From approximate membership checks to approximate state machines. In *SIGCOMM*, 2006.

[16] F. Bonomi, M. Mitzenmacher, R. Panigraphy, S. Singh, and G. Varghese. Bloom filters via d-left hashing and dynamic bit reassignment extended abstract. In *Forty-Fourth Annual Allerton Conf., Illinois, USA*, pages 877–883, 2006.

[17] P. Cheng, F. Ren, R. Shu, and C. Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data centers. In *NSDI*, 2014.

[18] Cisco. Netflow performance analysis. *White paper*, 2005.

[19] 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 *ACM SIGOPS*, 2007.

[20] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *ACM SIGCOMM*, 2003.

[21] D. Eppstein, M. Goodrich, F. Uyeda, and G. Varghese. What's the difference? efficient set difference without prior context. In *SIGCOMM*, 2011.

[22] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better netflow. *ACM SIGCOMM*, 2004.

[23] C. Estan and G. Varghese. Data streaming in computer networking. In *Workshop on Management and Processing of Data Streams*, 2003.

[24] M. T. Goodrich and M. Mitzenmacher. Invertible bloom lookup tables. In *arXiv:1101.2245v2*, 2011.

[25] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM*, 2015.

[26] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, 2014.

[27] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2), 2007.

[28] R. Kompella, K. Levchenko, A. Snoeren, and G. Varghese. Every microsecond counts: Tracking fine-grain latencies with a loss difference aggregator. In *SIGCOMM*, 2009.

[29] A. Kuzmanovic and E. W. Knightly. Low-rate tcp-targeted denial of service attacks (the shrew vs. the mice and elephants). In *SIGCOMM*, 2003.

[30] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 165–176, New York, NY, USA, 2006. ACM.

[31] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. In *SIGCOMM Comput. Commun. Rev.*, 1997.

[32] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38(2), 2008.

[33] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Algorithms — ESA 2001. Lecture Notes in Computer Science 2161*.

[34] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *SIGCOMM*, 2014.

[35] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

[36] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended Bloom filter: An aid to network processing. In *SIGCOMM*, 2005.

[37] F. Uyeda, L. Foschini, F. Baker, S. Suri, and G. Varghese. Efficiently Measuring Bandwidth at All Time Scales. In *NSDI*, 2011.

[38] B. Vöcking. How asymmetry helps load balancing. *J. ACM*, 50(4), 2003.

[39] W. Vogels. Performance and scalability. `http://www.allthingsdistributed.com/2006/04/performance_and_scalability.html`, 2009.

[40] M. Wang, B. Li, and Z. Li. sflow: Towards resource-efficient and agile service federation in service overlay networks. *Distributed Computing Systems, International Conference on*, 0:628–635, 2004.

[41] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *NSDI*, 2011.

[42] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI*, 2013.

[43] D. Zhou, B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Scaling up clustered network appliances with ScaleBricks. In *SIGCOMM*, 2015.

[44] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In *SIGCOMM*, 2015.

Sibyl: A Practical Internet Route Oracle

Ítalo Cunha[†*] *Pietro Marchetta*[‡*] *Matt Calder*[★] *Yi-Ching Chiu*[★]
Brandon Schlinker[★] *Bruno V. A. Machado*[†] *Antonio Pescapè*[‡]
Vasileios Giotsas[◇] *Harsha V. Madhyastha*[+] *Ethan Katz-Bassett*[★]

[†]Universidade Federal de Minas Gerais [‡]University of Napoli "Federico II"

[★]University of Southern California [◇]University of California San Diego/CAIDA [+]University of Michigan

Abstract

Network operators measure Internet routes to troubleshoot problems, and researchers measure routes to characterize the Internet. However, they still rely on decades-old tools like traceroute, BGP route collectors, and Looking Glasses, all of which permit only a single query about Internet routes—what is the path from here to there? This limited interface complicates answering queries about routes such as *"find routes traversing the Level3/AT&T peering in Atlanta,"* to understand the scope of a reported problem there.

This paper presents *Sibyl*, a system that takes rich queries that researchers and operators express as regular expressions, then issues and returns traceroutes that match even if it has never measured a matching path in the past. *Sibyl* achieves this goal in three steps. First, to maximize its coverage of Internet routing, *Sibyl* integrates together diverse sets of traceroute vantage points that provide complementary views, measuring from thousands of networks in total. Second, because users may not know which measurements will traverse paths of interest, and because vantage point resource constraints keep *Sibyl* from tracing to all destinations from all sources, *Sibyl* uses historical measurements to predict which new ones are likely to match a query. Finally, based on these predictions, *Sibyl* optimizes across concurrent queries to decide which measurements to issue given resource constraints. We show that *Sibyl* provides researchers and operators with the routing information they need—in fact, it matches 76% of the queries that it could match if an oracle told it which measurements to issue.

1 Introduction

Operators and researchers need Internet route measurements to keep the Internet running smoothly, to understand its behavior, and to improve it for the future [61, 75]. Route measurements help identify performance problems caused by circuitous routing [34, 58, 73], loops and loss caused by inconsistency during route convergence [11, 23, 28, 35, 36, 52, 60, 69], and outages caused by misconfigurations [7, 31, 32, 51, 74]. Route measurements can reveal malicious hijacks [76] and inadvertent routing leaks [24]. Route measurements are also used to understand the Internet's structure [2, 6, 29, 41, 57, 71] and performance [42].

The ideal: An Internet route oracle. Given the importance of route measurements, one can imagine a centralized platform that could be queried for any Internet route of interest. Which end-points in Europe route to each other circuitously via networks in other continents? Which routes traverse the Atlanta peering between Level3 and AT&T that seems to be experiencing congestion? Is the problem more widespread—which routes traverse a peering between Level3 and AT&T that is not in Atlanta? Which routes go through Level3 in Atlanta without going through AT&T? Which Tor exit nodes have routes to my destination that do not traverse the US? A platform that can answer such questions would enable better understanding and faster troubleshooting for researchers and operators.

The reality: Traceroute. While such a platform would be enormously useful, the reality today is far from it. We are stuck with tools like traceroute. While traceroute is simple, widely used, and has been incredibly useful [1, 2, 6, 7, 13, 22, 27, 29, 31, 32, 41, 42, 51, 57, 61, 71, 73, 74, 75, 76], it offers a very limited capability–it can only answer *"what is the path from here to there?"* We are used to asking this question, so it seems natural, but in fact it is only one of the many questions we might ask about Internet routes, limiting the ability of operators and researchers to access the routing information they need. The Outages network operators mailing list [51] illustrates the problem—operators frequently send a traceroute to the mailing list when experiencing problems [7],

[*] The two lead authors are listed alphabetically. They conducted some of this research as visiting scholars at USC.

asking other operators to send traceroutes from their vantage points, in the blind (and often unsuccessful) hope that someone will issue a measurement that illuminates the problem.

Our contribution: A practical traceroute-based oracle. While a complete oracle for Internet routing is clearly infeasible without radical changes to the network, we demonstrate that we can come surprisingly close using only available vantage points and measurement tools.

We present *Sibyl*,[1] our system that can serve a rich set of queries about Internet routes. *Sibyl*'s interface is simple yet powerful: a user submits a regular expression describing paths of interest (§3.2), and *Sibyl* returns routes that match. Users need not worry about which vantage points to use, how to access or configure them, or which destinations to target. Behind the scenes, *Sibyl* issues traceroutes from a diverse set of vantage points, with the goal of satisfying a query if any vantage point has routes that match. Our evaluation in Section 8.2 shows that combining vantage points from multiple measurement platforms achieves unprecedented coverage, better than even successful crowd-sourced measurements [55, 56].

The problem: Resource constraints limit measurement budgets. Although the integration of multiple sets of vantage points offers the *potential* to improve coverage [8, 64], most vantage points are severely constrained in the number of measurements they can issue. This constraint occurs because the most diverse sets of vantage points are in home networks [55, 56, 57, 62], on personal phones [73], and on production devices [63], settings in which measurements cannot be allowed to interfere with other uses of already scarce resources. Thus, exhaustive probing to answer a query is infeasible, and allocating limited measurements to maintain an up-to-date atlas in the face of path changes is an extremely hard problem [15].

The main challenge in building *Sibyl* to serve any query is that, due to its limited probing budget, it may have never previously measured a path that matches the query or, even if it did, the path may have changed since. As a result, it needs to serve queries despite uncertainty about which measurements match the queries.

Our approach: Allocate measurement budget based on predictions. Our primary technical contributions to overcome this challenge are three-fold. First, we demonstrate how *Sibyl* can use the structure of a query to focus its attention on a small number of traceroutes to consider issuing (§6). Second, we design a prediction engine which uses an atlas of previously-issued traceroutes to predict which unissued traceroutes are likely to match input queries (§5). Third, we develop an optimization framework that uses the predictions to allocate *Sibyl*'s probing budget to measurements that maximize how well it satisfies input queries (§4).

Building an effective prediction engine requires addressing potential causes of inaccuracy. First, the prediction engine could make an incorrect prediction from even an up-to-date atlas, due to inaccuracies in the modeling of routing policy. Second, measurements in the atlas may become out-of-date. So, we develop techniques to evaluate how likely a prediction is to be correct (§5.2), allowing *Sibyl* to incorporate the likelihood into its optimizations, and we develop lightweight approaches *Sibyl* can use to identify and patch or discard paths that may no longer be correct (§7).

Our evaluation (§8) shows that, using this prediction approach, *Sibyl* can serve 32% more queries than it could without calculating likelihoods and can, despite stringent rate limits, serve 76% of the test queries that it could if it had an oracle informing it which measurements to issue.

2 Motivating *Sibyl*'s approach

Traceroute is widely supported, and when the right traceroute measurement is at hand, it can prove useful for a range of tasks. Therefore, we use traceroute measurements as the basis of *Sibyl* and strive to overcome its limitations.

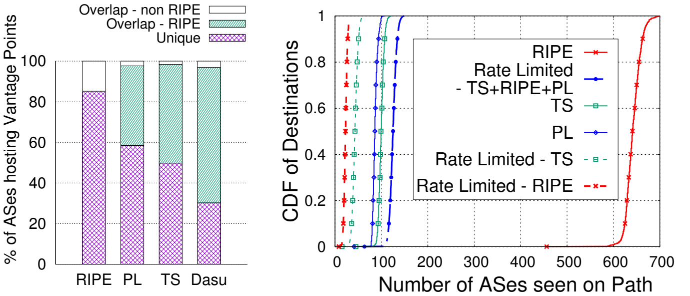
Opportunity: Combining platforms improves coverage. Today, one can use a number of publicly accessible measurement platforms that offer vantage points (VPs) across the world in order to issue traceroute measurements. In this paper, we focus on platforms at two extremes—small numbers of powerful VPs in a somewhat homogeneous deployment (PlanetLab) versus large numbers of severely limited VPs in networks around the world (RIPE Atlas and traceroute servers). In addition, Dasu and DIMES each offer several hundreds to several thousands of VPs from which one can issue traceroutes. For a few of these platforms, Figure 1a presents the number of VPs they offer and the number of ASes across which these VPs are spread. Although Figure 1a shows that the number of ASes in which RIPE Atlas offers VPs is much higher than in other platforms, we see in the *Unique* portions of the bars of Figure 1b that each of the other platforms contributes significantly to improving the number of distinct ASes covered by VPs. For all three of PlanetLab, Dasu, and traceroute servers, 30%–60% of ASes in which they have VPs do not host VPs for any of the other platforms.

Challenge: Resource constraints limit probing rates. The wide spread of RIPE Atlas and the presence of other VPs in ASes without Atlas probes show promising coverage for a unified system. But, effective use is compli-

[1] Named for the oracular **Siby**ls of ancient Greece, not the pesky **Sybi**ls who keep undermining our P2P systems.

Platform	# of VPs	# of ASes
PlanetLab (PL)	422	260
RIPE Atlas (RIPE)	7699	2716
Traceroute servers (TS)	499	494
DIMES	<400	–
Dasu	–	288

(a) Sizes of measurement platforms.



(b) Vantage point uniqueness. **(c)** Path diversity. Dashed lines obey rate limits.

Figure 1: Utility of combining measurement platforms: (a) Comparison of deployment sizes. (b) % of ASes hosting a platform's vantage points (VPs) that: (i) only host that platform; (ii) also host a RIPE Atlas VP; (iii) or host a VP from another non-Atlas platform. (c) Path diversity, with/without rate limits that exist in practice.

cated because the most diverse sets of VPs have severe and inevitable resource constraints. Good visibility requires an ability to measure from many networks low in the AS hierarchy [48], and researchers have argued and demonstrated that the way to achieve this viewpoint—especially in remote and developing regions—is to gather measurements from mobile devices [70, 73] and home networks [12, 17, 22, 55, 56, 57, 62], settings in which resources are scarce and researchers are guests. To give a sense of the constraints, measuring a traceroute every 5 minutes to all 500,000 BGP prefixes would take more than 40 Mbps—much higher than typical uplink bandwidth in many parts of the world.[2] And, to avoid interfering with the hosts, the platforms limit measurements to only a small fraction of this rate. Traceroute servers also offer diverse VPs, but these machines serve an operational role and so do not allow a fast rate of measurement. Future faster rates will still strain in the face of measurement-hungry use cases such as network tomography [9, 14] and studying route convergence [36, 69], which require consistent snapshots or rapid tracking of changes, respectively.

Figure 1c depicts one measure of the impact of limited probing budgets. It plots the number of unique ASes seen when using the vantage points in PlanetLab, RIPE Atlas, and traceroute servers in isolation and in combination, with and without rate limits. In each case, we consider traceroutes to the .1 address in the same 1000 IP prefixes, and we do not count the source AS (which we accounted for in Figure 1b). We have measurements from every PlanetLab site, RIPE vantage points in 2000 ASes not covered by PlanetLab, and traceroute servers in 200 ASes not covered by RIPE. Ignoring rate limits, RIPE vantage points provide routes to most destinations through > 600 transit ASes, versus only ≈ 100 when using PlanetLab or traceroute servers alone. Figure 1c also depicts the path diversity we can uncover if we allocate a day's Atlas probing budget and a day's traceroute server rate limit evenly across the 1000 destinations. RIPE enforces a per user aggregate rate limit across all sources and destinations. Here, we split it across a quarter of the Atlas VPs and 0.2% of the Internet's prefixes. We follow established research best practices [40, 59] and limit ourselves to one traceroute every 5 minutes per public traceroute server. These limits result in us randomly choosing 16–17 RIPE vantage points and 57–58 traceroute servers from which to probe each destination, and the graph shows results averaged across 10 trials. Given these severe rate restrictions, the benefit of PlanetLab—and its very high achievable probe rate—becomes clear, and the route diversity is much better if we combine the rate-limited traceroute servers and Atlas platform with the smaller, but less restrictive PlanetLab platform.

Challenge: Rate limits necessitate decisions in the face of uncertainty. The vast gap between the full diversity of paths seen in Figure 1c and the diversity seen when subject to rate limits shows that we have to be quite discerning in how we allocate a limited probing budget, to make sure we are issuing the measurements most useful to the queries at hand. We cannot issue measurements fast enough to have up-to-date paths to large numbers of destinations—the rate limits imposed by Atlas and traceroute servers [40, 59] mean that it would take years to measure routes from their VPs to all 500K BGP prefixes. Therefore, to serve queries well, it is necessary to reason effectively about which traceroutes to issue despite uncertainty about routes that measurements will traverse and, hence, which traceroutes will satisfy queries.

3 System overview

Goal. Our goal is to provide researchers and operators with route measurements of interest to them. Our system should allow them to express properties of interest in a natural way, without the user needing to know a

[2] Beyond constraints on the VPs, we do not want to overload upstream devices or links with measurement traffic, and routers and other devices increasingly rate-limit and de-preference these probes.

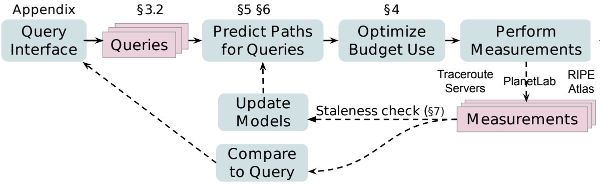


Figure 2: Sibyl architecture.

priori which (source, destination) pairs will yield paths with those properties. Section 2 shows existing traceroute platforms offer rich path diversity, if the system can respect resource constraints while efficiently measuring only the paths most useful in serving received queries.

3.1 Basic architecture

Figure 2 depicts *Sibyl*. Users submit queries to the system (§3.2). It operates in rounds, queueing up queries in between rounds. This round-by-round operation allows us to formulate the decision as a clean optimization, simplifies rate limiting, and aids in efficient use of a probing budget by batching requests. RIPE Atlas, for instance, is designed to perform more efficiently when given batches of measurements. Each round, the system predicts which traceroutes might be useful to match pending queries (§5 and §6). It then formulates an optimization to select the traceroutes to issue given measurement resource constraints (§4.1–4.2) and solves the optimization greedily (§4.3). It issues the measurements, collects the results, and returns results that match user queries.

Sibyl currently uses vantage points from PlanetLab, RIPE Atlas, and traceroute servers. We developed a controller for each that exposes them via a common API, including information on available vantage points and rate limits, and commands to request and collect traceroutes. A central controller integrates the three platform controllers to present the rest of *Sibyl* with a unified view. In the background, we issue daily traceroutes from all PlanetLab sites to responsive destinations across the Internet [25] to bootstrap *Sibyl*'s knowledge of routing.

3.2 Specifying queries

Just as other work found regular expressions to be a natural way to express properties of paths [46, 67], we support queries in a form that we refer to as symbolic regular expressions over IP addresses. Symbolic regular expressions are an analogue to symbolic finite automata [65], in which transitions are labeled with Boolean predicates on IP addresses, rather than directly with IP addresses. These predicates allow, for example, the natural expression of `Sprint(x)&CHI(x)` rather than listing all Sprint IP addresses in Chicago. We will use the notation `Sprint&CHI`. A predicate can delineate any subset of IP addresses, but our UI currently supports ASes, cities, and countries, [3] and also prefixes for sources/destinations.[4]

Users augment their queries with a utility function that indicates how well a set of traceroutes satisfies their needs. *Sibyl*'s UI currently supports two types of utility functions that we believe cover a wide range of queries. For *existence queries*, the user wants one matching path, e.g., the user may want to know the path from a particular network to a specific destination. The utility is zero if no measurements match, or a constant value if one or more measurements match. For *diversity queries*, the user wants a set of paths matching the query in as diverse ways as possible. For example, the user may want to know all paths that pass through a given AS link, in order to learn the set of (source, destination) pairs that use that link. The utility is a function of path diversity, which we model as a constant times the number of distinct elements seen in the set of measurements that match the query. The user specifies the granularity of elements by selecting any combination of (AS, city, and country). Again, if none of the traceroutes match the query, the utility is zero.

Now, let us consider a few example queries, in POSIX ERE-like syntax with dashes in between symbols for clarity. Parentheses create a group, whereas curly braces indicate that the query is a diversity query and delineate the portion of the query to diversify over.

Reverse traceroute [30]: To query for a path from a network *r* back to a source *s*, the user requests:

```
r-.*-s$
```

Detecting prefix hijacks with iSpy [76]: iSpy monitors paths towards a prefix *p* in the background. When the AS loses reachability to other destinations, iSpy considers it a normal outage if the destinations share common subpaths to the AS, or a hijack if the destinations represent a large cut in the graph towards *p*. To identify diverse AS paths for iSpy to monitor, an operator could query for:

```
^{.*}-p$ by <AS>
```

Troubleshooting a problem [51]: On January 6, 2015, an operator emailed the Outages mailing list suspecting a problem on paths that went between Level3 in LA and GTT in Seattle, and he wanted to check other paths with that subpath. He was requesting:

```
^{.*-(GTT&SEA-.*-Level3&LAX |
Level3&LAX-.*-GTT&SEA)-.*}$ by <AS,city>
```

[3] Mapping IP addresses to PoPs, ASes, and locations are active areas of research. We use iPlane's PoP and AS mappings and MaxMind's location data. *Sibyl* is agnostic to how mappings are generated, and its results will improve as mappings do.

[4] Our techniques for deciding which measurements to issue in response to a query (§5) base decisions on previous measurements of routing, so implicitly encode routing policies and hence avoid wasting measurements trying to match unlikely regular expressions, such as one that asks for a path that traverses every Tier-1 network.

Another operator replied that traceroutes with the problem seemed to traverse a Seattle peering between GTT and NTT. To see if the problems occurred on GTT paths with other peers as well, one might query for:

```
^.*-{[^NTT-Level3]}-GTT&SEA-
      {[^NTT-Level3]}-.*$ by <AS>
```

Appendix I presents screenshots of *Sibyl*'s query interface for the last of these examples.

3.3 Key problems to solve

Section 2 described two key challenges *Sibyl* must overcome in integrating traceroute platforms to serve these types of queries: severe probing rate limits induced by resource constraints, and the need to decide how to allocate these limited probes despite not knowing definitively which traceroutes will satisfy queries. To overcome these challenges, we address the following sub-problems:

- (§4) Suppose we can address uncertainty by capturing the likelihood that a traceroute, if issued, will match a query. How should *Sibyl* allocate its probing budget to best serve queries?
- (§5) How can *Sibyl* calculate those likelihoods?
- (§6) Given that the set of possible measurements is large, how can *Sibyl* limit the set of traceroutes it has to consider issuing (and hence calculate likelihoods for)?

4 Maximizing returns from rate limits

Since *Sibyl* cannot issue every traceroute—or even every traceroute that would match the queries—in a given round, it needs to intelligently allocate its probing budget to best serve a set of queries. Because *Sibyl* must issue a traceroute in order to know definitively whether it matches a query, *Sibyl*'s goal is to maximize the expected utility of the traceroutes it issues. This section describes how *Sibyl* allocates its budget, assuming it has an oracle that answers, for every possible traceroute, the likelihood that the traceroute, if issued, will match a particular query. Section 5 describes how *Sibyl* estimates these likelihood values to approximate such an oracle.

4.1 Accounting for rate limits

Since we want *Sibyl* to incorporate different sets of VPs to improve coverage and path diversity, we need to account for the different *kinds* of rate limits across platforms. The rate at which a PlanetLab node can probe is limited by the ability of our traceroute tool to send and receive and by the available bandwidth. ISPs make traceroute servers available through websites, but restrict how often one can issue traceroutes from a website. RIPE Atlas users earn credits for hosting probes, then spend credits by issuing measurements. We host a number of Atlas probes in order to earn credits, but RIPE caps the number of credits a user can spend in a day regardless of credit balance.

We unify these different types of rate limits as follows. First, we group together each set of vantage points that are subject to a shared aggregate rate limit. For the i'th such set, we will use the notation $V_i = \{v_{i,1}, v_{i,2}, v_{i,3}, \ldots\}$ to indicate the vantage points in set V_i, and let $\mathcal{V} = \{V_1, V_2, V_3, \ldots, V_n\}$ be the collection of n sets used. For PlanetLab, each host is in a singleton set, since the number of traceroutes sent from one PlanetLab site does not affect the number that can be sent from another. For traceroute servers, we group the hosts behind a common web interface (generally the hosts in one ISP), since we are limited in how often we can query a website without drawing complaints. For RIPE Atlas, we group together all vantage points in the platform, since they are subject to a platform-wide credit budget and daily limit.

Second, in each round, *Sibyl* has a multi-element budget of traceroutes it can issue, with one budget per set of vantage points in $\mathcal{V}$. For rate-limited vantage points like PlanetLab or traceroute servers, the per-round traceroute budget for a set reflects the rate limit on the set and the duration of the round. For credit-based vantage point platforms like RIPE Atlas, we set a per-round aggregate budget for all traceroutes from the platform to reflect the number of credits we earn in a round.[5]

4.2 Formulating the optimization

In a given round r, we have a set of queries $Q = \{q_1, q_2, \ldots, q_m\}$, each with a corresponding utility function $f_{q_1}, f_{q_2}, \ldots, f_{q_m}$ that maps a set of traceroutes to a score. For each set of vantage points $V \in \mathcal{V}$, we have a per round budget C_V. Each V defines a set of possible traceroutes $T_V = \{t_{v,d} \mid v \in V, d \in$ the set of all Internet destinations $D\}$, where $t_{v,d}$ is the traceroute from v to d, and we have to select a subset $T_{r,V} \subseteq T_V$ to issue in round r such that $|T_{r,V}| \leq C_V$.

Our goal is to select traceroutes, subject to budget constraints, to maximize the combined utility across queries:

$$\begin{aligned} &\max_{T_r} f(T_r), \text{ where } T_r = \bigcup_{V \in \mathcal{V}} T_{r,V} \\ &\qquad \text{and } f(T_r) = \sum_{q \in Q} f_q(T_r) \qquad (1) \\ &\qquad \text{subject to } |T_{r,V}| \leq C_V \quad \forall V \in \mathcal{V} \end{aligned}$$

Since we cannot know whether a traceroute satisfies a query before issuing it, in practice, *Sibyl* maximizes the

[5] We adjust the exact budget round-by-round to allow overspending when we have banked a surplus or exercise caution when reserves run low, as well as to cap it to not exceed the daily platform limit.

expected utility. We use the notation $t \in q$ to indicate that traceroute t satisfies query q. Assuming an ability to determine the likelihood $p(t \in q_{\text{exist}})$ for any traceroute t matching an existence query q_{exist},[6] *Sibyl* calculates the expected utility $\mathbb{E}\left[f_{q_{\text{exist}}}(T_r)\right]$ as the probability that at least one traceroute matches the query:

$$\mathbb{E}\left[f_{q_{\text{exist}}}(T_r)\right] = 1 - \prod_{t \in T_r} (1 - p(t \in q_{\text{exist}})) \qquad (2)$$

Sibyl calculates the expected utility for a diversity query in a similar way, except that the likelihood values capture, for every path element h at the diversification granularity as defined by a Boolean predicate on IP addresses, the probability that t satisfies q_{div} *and* traverses h (Appendix D.3 presents an example):

$$\mathbb{E}\left[f_{q_{\text{div}}}(T_r)\right] = \sum_{h} \left(1 - \prod_{t \in T_r} (1 - p(t \in q_{\text{div}} \wedge \exists i \in t : h(i))\right) \qquad (3)$$

(In practice, *Sibyl* scales down diversity utility scores, which are per (query,hop), to balance vs existence queries, which are per (query).)

4.3 Solving the optimization

We apply a greedy algorithm to select the measurements to issue in every round. At each step, *Sibyl* chooses to issue the traceroute that fits in the budget (meaning that the source VP must be part of a set V for which budget remains) and that provides the largest marginal expected utility on top of those already chosen.[7] It stops when no budget remains for the round or when no traceroutes provide additional expected utility. While it may seem like a complicated problem, with a multi-part budget, multiple queries, and queries that desire diverse sets of traceroutes, in fact the greedy algorithm is known to have a provably good approximation bound for this class of problems. See Appendix A for details.

In addition to having good approximation performance, the runtime of our greedy algorithm is reasonable. The runtime is reasonable because both existence and diversity queries allow us to calculate the marginal expected benefit of each possible traceroute in time proportional to the number of queries, without growing with the number of traceroutes already issued. Appendix A describes other utility functions *Sibyl* supports. The worst-case runtime is thus proportional to the size of the budget (the number of greedy steps) times the number of traceroutes under consideration (to assess marginal benefit of each during each greedy step) times the number of queries (to calculate the marginal utility of the traceroute). Most traceroutes under consideration do not match most queries, i.e., $f_q(t) = 0$ most of the time, simplifying the calculation of marginal benefit in practice. *Sibyl* also limits the number of traceroutes under consideration based on the structure of queries (§6).

5 Estimating likelihood of satisfying queries

Sibyl approximates an oracle by using the subset of paths for which it has relatively fresh measurements to predict other paths, checking whether the predictions match queries, and estimating how confident it is in the predictions. PlanetLab paths are stable relative to how often we can refresh PlanetLab measurements. Further, while paths from diverse RIPE Atlas and traceroute server VPs in general change more than PlanetLab paths and cannot be refreshed frequently, the portions of paths near these VPs tend to be quite stable.[8] Based on these observations, our design predicts paths by composing the relative freshness of paths to destinations from PlanetLab with the long-term stability of the beginning portions of paths from other VPs in order to predict unknown paths from these VPs, overcoming the rate limits that keeps us from measuring a full map in a timely fashion.

5.1 Predicting unknown paths

We adapt iPlane's path splicing approach [39] to predict whether a particular unmeasured path is likely to match a query. To predict the path from s to d, iPlane splices a path from s (to some destination) with a path to d (from some source), if they traverse a common point of presence (PoP, a set of routers in the same location and same AS), which we refer to as the splice PoP.

Although iPlane's approach provides a basic mechanism for using measured paths to predict unknown paths, it has two major limitations for our needs. First, iPlane's predictions can be wrong; our experiments found 32% of its AS path predictions to be incorrect. Second, iPlane does not calculate how confident it is in its prediction. Even if iPlane predicts (vantage point, destination) pairs as candidates to match a query, it fails to provide guidance on which paths are more likely to match the query than others, given limited measurement budgets.

We overcome these shortcomings in iPlane's path splicing approach as follows. For a (vantage point v, destination d) pair, while iPlane selects a single best guess for the route between them, we instead consider all possible ways to splice previously measured paths from v with previously measured paths to d. We then estimate

[6] For simplicity, we assume independence in how well traceroutes satisfy different queries, and in whether different traceroutes satisfy a query.

[7] The marginal expected utility of adding a traceroute t to a set of previously selected traceroutes T is $\mathbb{E}[f(T \cup \{t\})] - \mathbb{E}[f(T)]$.

[8] Measurements supporting these claims appear in Appendix B.

our confidence in the correctness of each spliced path in the set S of all possible spliced paths from v to d. We denote the confidence as $p(v \rightarrow d = s)$ (normalized such that $\sum_{s \in S} p(v \rightarrow d = s) \leq 1$). Given these confidence estimates, we compute the likelihood $p(v \rightarrow d \in q)$ of the traceroute from v to d matching a query q as the sum of confidence in the spliced paths that match the query:[9]

$$p(v \rightarrow d \in q) = \sum_{s \in S \wedge s \in q} p(v \rightarrow d = s) \tag{4}$$

Appendix D.2 illustrates how the above process works.

5.2 Assigning confidence to predictions

To assess the confidence in each spliced path, we employ RuleFit, a supervised machine learning technique [20]. We describe how we train and apply a RuleFit model. The model takes the set of spliced paths of a particular prediction and assigns confidence to each based on features of the paths.

Training the RuleFit model RuleFit is a supervised machine learning technique based on rule ensembles. In our case, we supply RuleFit with a training set that maps from features of a predicted (spliced) path (e.g., the predicted path's AS-path length and the latency from the source to the splice point; Appendix C describes all features) to the similarity between the spliced path and the actual path. As a measure of similarity, we use the PoP-level Jaccard index. RuleFit then generates thousands of rules that combine features in logical expressions and builds a model using rules that help predict the Jaccard index. RuleFit automatically generates and selects rules (and indirectly, features) using techniques such as decision trees and lasso constraints. See the RuleFit paper for details [20]. Each rule selected by RuleFit has an associated value, with positive (negative) values for rules meant to identify predicted paths similar (dissimilar) to the real path, indicating high (low) Jaccard index. Important features may change over time, as the Internet and the set of *Sibyl* VPs evolves. We track the accuracy of predictions over time to identify if performance drops and can re-initiate training. For the evaluation in Section 8, we use traceroutes from 100 PlanetLab sites to 500 destinations and from all RIPE Atlas (Atlas) and traceroute server (TS) sites to 50 destinations to generate spliced paths from Atlas and TS sites to the 500 destinations. We randomly chose 2.5% of the spliced paths to train a RuleFit model.

Using the RuleFit model To use the model to estimate the Jaccard index of a predicted (spliced) path from v to d, *Sibyl* calculates the features of the predicted path, then uses the RuleFit model to score the path. The score for a spliced path is the sum of rule values for rules that match the spliced path's features; e.g., if the spliced path's AS-path length is among the shortest, then increase the confidence (score) that it is very similar to the actual path. It repeats this process for every spliced path between (v, d).

Sibyl translates these estimates of similarity between a (known) spliced path and (unknown) actual path into a confidence estimate that a query that matches (does not match) the spliced path will also match (not match) the actual path. We assign each predicted path a confidence proportional to its RuleFit score, normalized to sum to the highest predicted Jaccard index among all spliced paths for $v \rightarrow d$. *Sibyl* uses these confidence values to estimate the likelihood that a traceroute will match a query, using Eq. 4, which it then uses to optimize the expected utility of the traceroutes it chooses to issue, in Eq. 1.

Section 8.3 evaluates the accuracy of *Sibyl*'s likelihood estimates, Appendix E evaluates the accuracy of its Jaccard index predictions, and Appendix C describes the RuleFit model in more detail.

6 Limiting traceroutes to consider

Thus far, our description has assumed that we estimate the likelihood of matching a query for every possible traceroute from every vantage point, and then use these likelihood values to choose the subset of traceroutes that *Sibyl* should measure in order to maximize utility, given rate limits. However, due to the non-negligible computation associated with the estimation of likelihood values, running this computation on all (vantage point, destination) pairs is not practical.

Instead, *Sibyl* computes the likelihood of matching a query q only on a subset of candidate paths it deems likely to match the query. The goal of candidate generation is to identify (vantage point v, splice PoP r, destination d) tuples such that *Sibyl* has a previous traceroute from v going through r that matches a prefix of the query q (possibly the empty prefix), and has a traceroute to d through r that matches the remaining suffix (possibly empty). For example, candidate generation for the query `Level3-Cogent-.*-SmallISP` could find a path that traverses a `Level3-Cogent` link on the way to some r, then another path that traverses r on its way to `SmallISP`. The process works as follows.

1. Given the query q, construct a symbolic finite automaton A_q that accepts L_q, the language of paths that match the expression q.

[9] The likelihood value for an (v, d) pair need not directly or inversely correlate with the number of spliced paths between the pair, as it depends on how confidence varies across spliced paths and on which spliced paths match the query. For example, if $p(v \rightarrow d = s_1) = 0.5$ and $p(v \rightarrow d = s_2) = 0.25$, $p(v \rightarrow d \in q)$ could be 0.25 ($s_2 \in q$), 0.5 ($s_1 \in q$), 0.75 ($s_1 \in q$ and $s_2 \in q$), or 0 (neither matches).

2. Run A_q over all traceroutes previously gathered from *Sibyl*'s VPs, which consists of evaluating the hops' IP addresses against A_q's transition predicates, testing, for example, AS membership. Label each (source, PoP) tuple with all of the state-to-state transitions that A_q can follow in consuming one of the PoP's IP addresses when processing a traceroute from that source.
3. Build A_q^R by swapping A_q's initial and final states and reversing transitions. A_q^R accepts the language L_q^R consisting of the reverse of all paths in L_q.
4. Run A_q^R over all traceroutes, starting from the destinations and proceeding backwards, labeling each (destination d, PoP r) tuple with all the transitions that A_q^R can follow in consuming r starting from d.
5. If a PoP ends up labeled as following a transition in one direction from a source and in the opposite direction from a destination, then the spliced path matches the entire query.

Appendix D.1 presents an example of this sequence of steps.

7 Patching & pruning stale measurements

Previous sections assume the availability of an atlas of historical measurements that serve as the basis for predictions. Resource-constrained VPs do not have enough resources to refresh all measurements regularly, and so routes may change between measurements. Therefore, *Sibyl* needs to balance between discarding old measurements to reduce the risk of out-of-date ones causing faulty predictions, versus using them in predictions to aid coverage (since many old measurements may still be valid). Given that most (*s, d*) pairs use a single route the vast majority of the time [15, 52], we err on the side of retaining routes and apply three mechanisms to infer and remove stale data from *Sibyl* 's atlas. In the first two, a traceroute from *s* to *d* reveals a change in one path, and we use the new path to patch other paths either from *s* or to *d* that overlapped the old path from *s* to *d*.

Traceroute-based destination patching. Since Internet routing is destination-based, if two traceroutes to the same destination (possibly from different sources) converge, we patch the old measurement to match the new measurement from the convergence point to the destination. Flach et al. found that the most common reason for violations of destination-based routing is load balancing [18], which can be filtered using Paris traceroute [5, 66]. Excluding load balancing, that study found only 10% of routers caused IP-level deviations from destination-based routing and only 2% caused AS-level deviations, for reasons including traffic engineering and tunneling. In the future, we could apply these earlier techniques to identify and exclude these exceptions from our patching.

Traceroute-based source patching. Destination-based routing helps us keep the tail of paths collected from constrained VPs up-to-date using measurements from less-constrained VPs such as PlanetLab. To remove stale data from the beginning of paths, we assume a path change observed on the path from a constrained VP to one destination will also impact its paths to other destinations that traverse the path segment that changed. Violations to these assumptions result in incorrect updates to paths. However, a single error is unlikely to impact *Sibyl* 's predictions, as *Sibyl* can continue to make equivalent predictions if it knows other non-stale paths from the VP that traverse a subset of the PoPs that were on the stale segment.

BGP-based destination pruning. Traceroute-based patching still requires issuing a measurement to detect the change. We supplement these approaches with lightweight BGP monitoring, which requires only passive observation of BGP feeds via the following steps. First, we convert all traceroutes in the atlas into AS paths in the following process. (a) We use PeeringDB data to build a database of IXP prefixes and remove these IP addresses from traceroutes. (b) We map remaining IP addresses to the ASes that originate their prefixes. (c) We group addresses into routers using CAIDA's Midar [33] for IP aliasing resolution, assigning a router to an AS only if all its interfaces belong to the same AS. (d) We partition the traceroute into segments in which every router has been assigned an AS (but a segment can contain multiple ASes). Second, we monitor RouteViews and RIPE RIS BGP feeds for BGP changes. When we observe an AS *A* change its next hop AS to a destination *d*, we mark as stale any traceroutes that routed via *A* and its old path to reach *d*, and we do not use these traceroutes to make predictions. Whereas traceroute-based staleness checks provide a way to patch old measurements, BGP checks on their own do not.

8 Evaluation

We evaluate *Sibyl* from two perspectives. First, we show that *Sibyl* is able to serve queries effectively. On a large set of test queries, it satisfies three-quarters of the queries it could if it had an oracle to provide the result of a traceroute before issuing it. Thereafter, we evaluate individual components of the system in isolation to show that *Sibyl*'s components operate efficiently and make decisions that enable it to make good use of its probing budget.

8.1 Efficiency in serving queries

Datasets and experimental design. To evaluate *Sibyl* end-to-end, we run the system in an offline mode, stubbing out the component that issues traceroutes. We first collect a large set of traceroutes. We then run *Sibyl* as normal, except that, when it decides to issue a traceroute from a vantage point to a destination, instead of issuing a new measurement, it fetches the existing measurement between that pair. Offline analysis allows us to compare choices made by *Sibyl* with other measurements it chose not to issue or that we did not give it access to.

Between January 13–16 2016, we issued traceroutes from 2660 vantage points–2000 RIPE Atlas vantage points, 560 traceroute servers, and 100 PlanetLab sites–to 1000 destinations, chosen at random from a list known to be responsive [25]. Within a platform (Atlas, traceroute servers, or PlanetLab), each vantage point is in a different AS, although these is some overlap across platforms.[10]

In each experiment, we generate a starting corpus of paths that *Sibyl* has access to. The corpus includes all traceroutes from PlanetLab sites, giving it traceroutes to destinations to splice to for predictions. For rate limited platforms (traceroute servers and RIPE Atlas), the corpus starts with 10 randomly chosen measurements from each vantage point, a number previous work shows captures upstream diversity for path prediction [40].

The experiments test how efficiently *Sibyl* can allocate a limited number of additional traceroutes from rate-limited vantage points in order to serve queries. We emulate a series of rounds, with a per round measurement budget and query arrival rate configured per experiment and described with the experiments below. In each round, *Sibyl* decides how to allocate its probing budget to issue traceroutes, we assess how well these traceroutes matched the queries, and then we add these traceroutes to *Sibyl*'s corpus for the next round. Unsatisfied queries do not carry over to the next round.

Existence queries. We first evaluate *Sibyl*'s ability to serve *existence queries*, where the goal is to find one traceroute that matches. To generate test queries, we select one of the traceroutes not (yet) available to *Sibyl* and generate a query that will match it. This way, we know that there is at least one measurement that *Sibyl* could issue to match the query. To create a query, we sampled hops in the path to generate regular expressions according to four different Sibyl use cases (e.g., find paths that traverse a given link toward a destination; more details in Appendix F). We evaluated *Sibyl* with a range of budgets and query volumes, and the results are qualitatively similar, so we present results for just one setting, a per-round probing budget that allowed an average of one traceroute per query.

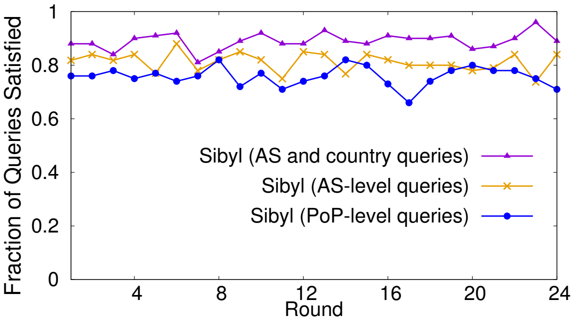


Figure 3: Fraction of queries satisfied when they are specified at different granularities.

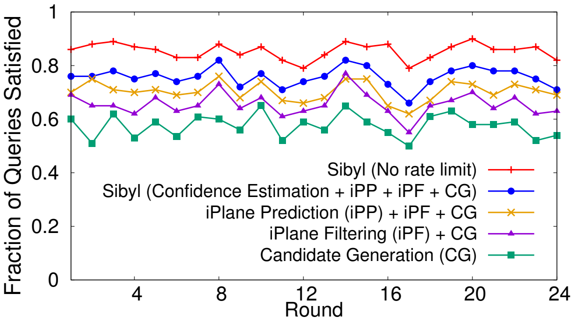


Figure 4: Incremental contribution of *Sibyl*'s components to its ability to satisfy existence queries. Combined, the techniques allow it to allocate budget smartly and approach the performance it would see if it could issue every candidate traceroute it generated (*no rate limit* line).

Performance by query granularity. We used this query generation approach at different granularities (by mapping the traceroutes to PoPs, ASes, and a mix of ASes and countries). Figure 3 shows the fraction of existence queries that *Sibyl* can satisfy at these granularities in each round. At all granularities, *Sibyl* satisfies a high fraction of queries. As expected, the coarser the granularity, the higher the fraction of satisfied queries, from around 75% at the PoP level to 90% at the AS/country level. Also, *Sibyl* is able to efficiently allocate its budget at different granularities, including answering queries that combine ASes and country codes, which may overlap in complex ways (e.g., traverse a link between AT&T and Level3 in the US on the way to Europe).

Incremental contribution of Sibyl components. *Sibyl*'s performance is good across queries of different granularities, and so we focus the rest of our analysis on fine-grained PoP-level queries to stress the system. For PoP-level queries, *Sibyl* allocates its probing budget well, satisfying 32% more queries than a baseline approach that relies only on existing measurements to answer queries. Figure 4 breaks down the incremental benefit of the sys-

[10] We worked with the RIPE Atlas staff to gather data faster than their normal rate limits. They allowed this just for the purpose of our evaluation, it required tight coordination between our team and theirs, and it does not appear they will support this on a regular basis.

tem's various modules. First, *candidate generation* uses *Sibyl*'s module that splices previously measured paths to identify (s,d) pairs that may match a query (§6). Without access to the rest of *Sibyl*, it then assumes that each of these pairs will indeed match the query and distributes measurements uniformly across queries, picking candidates at random for each query. Second, we add *iPlane filtering* to candidate generation, using iPlane to predict a (more accurate) PoP-level path for each candidate and filtering out candidates whose predicted paths do not match any query. Third, *iPlane prediction* extends iPlane filtering to consider all spliced paths iPlane can generate for a given candidate (s,d) pair (§5.1). Unlike the full system, this comparison point assigns an equal confidence value to every spliced path between the pair when calculating likelihood of matching a query. Finally, the *Sibyl* line adds in confidence (§5.2) into the likelihood estimation to arrive at the full system. As we add the techniques, each contributes to satisfying 5-8% additional queries, justifying their use.

Why does Sibyl fail to satisfy some queries? The two central challenges are (a) limited probing budgets and (b) uncertainty about whether traceroutes will match queries before issuing them. Because the evaluation uses a 1:1 query:budget ratio and all queries have at least one traceroute that satisfies them, without uncertainty, we could satisfy 100% of queries. *Sibyl* could miss satisfying a query either because it failed to generate any candidate traceroutes that, if issued, would satisfied the query, or because it did generate the candidate but calculated that it was unlikely to match the query. The *Sibyl (no rate limit)* teases apart these two causes, as it allows *Sibyl* to issue every candidate traceroute. Without a rate limit, *Sibyl* satisfies 88% of queries (vs. 76% with a 1:1 ratio), suggesting that half of *Sibyl*'s unsatisfied queries were because its corpus of measured paths did not suffice to generate candidates that could satisfy them, and half were instances in which *Sibyl* generated a candidate that would have satisfied the queries, but rated them as having less expected value than other candidates, so did not allocate probing budget to them. This result suggests the potential benefit of future work to improve candidate generation and likelihood estimation.

Can Sibyl efficiently service satisfiable queries in the face of unsatisfiable ones? Our evaluation thus far is on queries that are satisfiable, generated from traceroutes *Sibyl* could choose to issue. Appendix H presents an experiment demonstrating that the fraction of these queries that *Sibyl* can satisfy is robust to the simultaneous introduction of realistic but unsatisfiable queries.

Diversity queries. Next, we assess *Sibyl*'s ability to respond to *diversity queries* by finding a set of paths that match the query in diverse ways. We create diversity queries by supplementing the PoP-level queries from above by asking *Sibyl* to maximize diversity of all wildcard tokens (`.*`) in the regular expression. The diversity utility function for a query awards a unit of utility for each unique PoP in the set of matching traceroutes. We set a 1:4 query:budget ratio (but *Sibyl* may distribute this budget unevenly across queries according to its expected diversity optimization in Eq. 3).

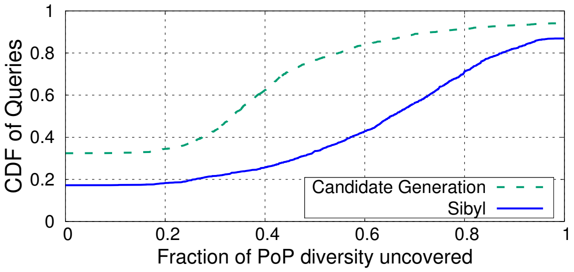


Figure 5: Fraction of distinct PoPs returned for diversity queries, relative to the number of distinct PoPs that could be returned if *Sibyl* had budget to issue every candidate traceroute it considered. Even with limited RIPE Atlas and traceroute server budgets, *Sibyl* usually uncovers a significant fraction of the relevant diversity, providing substantial benefit over an approach that lacks the ability to predict which paths are most likely to provide diversity in order to optimize use of the budget.

Figure 5 depicts the ratio between the number of distinct PoPs on matching paths that *Sibyl* uncovers subject to the rate limit versus the number it could have uncovered with an unlimited budget to probe all candidates it generates. The *candidate generation* baseline—which already uses some of *Sibyl*'s novel functionality to identify promising traceroutes to issue—is unable to find any matching paths for 32% of queries, and it uncovers less than half of the matching path diversity for 75% of queries. In contrast, by optimizing based on its estimation of the expected chance of a given traceroute traversing each PoP while satisfying the query, *Sibyl* satisfies 83% of queries with at least one traceroute, uncovers half the path diversity for 67% of queries, and, for 13% of queries, uses its very limited budget to uncover all of the diversity that was found using unlimited probes.

8.2 Coverage of vantage point platforms

Coverage by vantage point AS. Our ideal is to service *any* routing query, but *Sibyl* is limited by available vantage points. No one platform has achieved overwhelming coverage, and the types of ASes that host vantage points can vary across platforms, so we designed *Sibyl* to accommodate a range of platforms. Figure 6 depicts the locations of the vantage points of different platforms in terms of their coverage of ASes by customer cone sizes [3]; the customer cone of an AS is its customers, its customers' customers, etc.. For example, even though Atlas covers

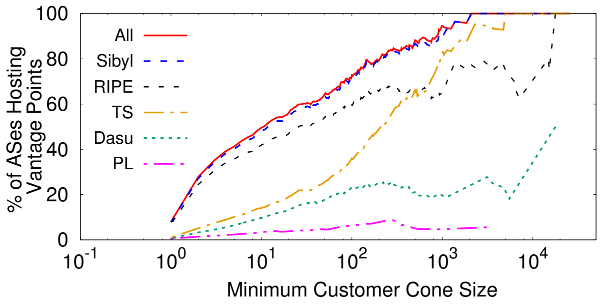


Figure 6: *Sibyl* combines platforms to maximize AS coverage, with especially strong coverage of larger ASes.

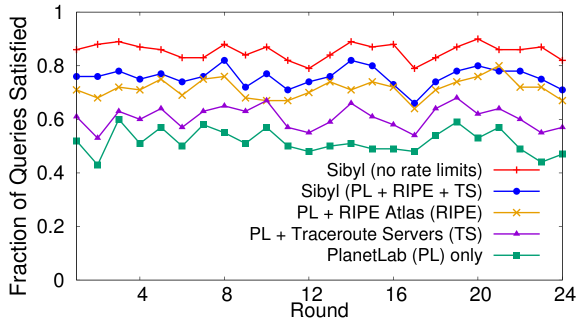


Figure 7: Fraction of existence queries satisfied given different VPs. Combining platforms improves performance.

by far the most ASes (Table 1a), the figure shows that traceroute servers have better presence in large ASes, enabling *Sibyl*'s union of platforms to have presence in all ASes with customer cone size greater than 2000, whereas this coverage is below 80% when combining all other platforms excluding traceroute servers. Based on data provided by the Dasu team, incorporating Dasu would not significantly improve *Sibyl*'s vantage point diversity, although it would increase probing budget. Coverage across AS sizes will improve as existing measurement platforms expand and new ones become available.

Impact of combining vantage point platforms on ability to satisfy queries. We now consider how combining platforms helps *Sibyl* satisfy queries. Using the same queries and probing budget as in Section 8.1, Figure 7 shows the fraction of queries *Sibyl* can satisfy using only PlanetLab, PlanetLab plus traceroute servers, PlanetLab plus RIPE, and all three platforms combined. The *Sibyl* lines from the Figure 4 are the same as the *Sibyl* lines in this graph. We observe that all platforms contribute to the number of satisfied queries. Even though the number of RIPE Atlas vantage points is four times larger than the number of traceroute servers, traceroute servers provide additional diversity and are useful in satisfying queries.

8.3 Accuracy of likelihood estimation

We evaluate *Sibyl*'s likelihood estimation (§5) during our end-to-end evaluation of existence queries (§8.1). Figure 8 shows the fraction of candidates that match a query as a function of estimated likelihood (Eq. (4)) for 20,895 candidates generated for 2273 queries. We bucketed candidates by rounding the estimated likelihood to the closest 0.1. The graph shows high correlation between likelihood and the probability of satisfying a query. Appendix E shows the number of candidates in each bucket.

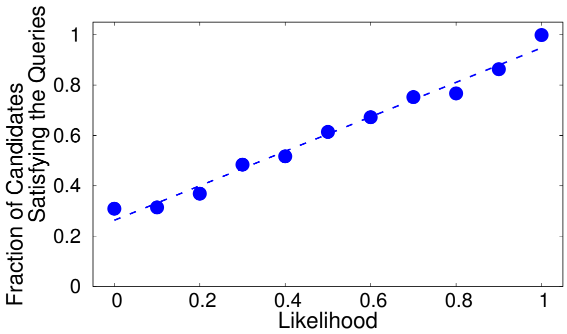


Figure 8: (Bucketed) estimated likelihood of candidates matching queries vs actual fraction matching queries.

8.4 Impact of staleness

Sibyl always issues and returns fresh traceroutes to serve queries, so staleness cannot result in false query matches. Staleness can however lead to wrong predictions and suboptimal allocation of probing budget.

We evaluate the impact of staleness on *Sibyl*'s ability to service queries over time, using weekly traceroute measurements from 1800 RIPE Atlas nodes toward a set of 1000 destinations collected between Jun. 20th and Aug. 20th, 2015. We partition the set of RIPE Atlas VPs in two: we choose 150 VPs at random to use as *constrained* VPs, and use the remaining 1650 VPs as *unconstrained* VPs. As in Section 8.1, we consider existence queries, give *Sibyl* a probing budget of one traceroute per query, and build an initial corpus of traceroute paths that includes 10 measurements from each of the 150 constrained VPs plus all measurements from the 1650 unconstrained VPs.

We test the performance of three different strategies for dealing with stale traceroutes. *Keep last 14 days* uses only paths collected during the last 14 days and discards older paths. *Keep all* accumulates all the traceroute paths collected by *Sibyl* regardless of their age, without applying any sanitation technique to mitigate staleness. *Sibyl(patching and pruning)* also accumulates all the collected traceroutes, but attempts to filter-out stale hops using *Sibyl*'s techniques described in Section 7.

Figure 9 measures *Sibyl*'s ability to service the queries over time. We also show linear fits for each curve to make the trends more clear. *Keep last 14 days* loses path diversity over time, as it only keeps traceroutes from parts of the Internet that were recently targeted by queries, and this narrow focus over time limits its ability to serve

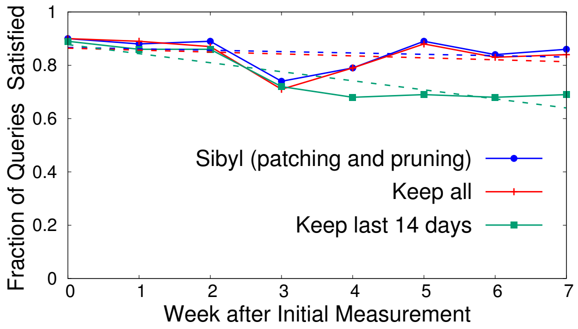


Figure 9: Queries satisfied over time using different approaches to maintain historical traceroutes as a basis for predictions. By pruning/patching paths it detects as stale, *Sibyl* performs much better than an approach that only keeps recent measurements and slightly better than an approach that keeps all measurements. The dip at week 3 is caused by corrupted traceroute files that week.

queries about some other parts of the Internet. *Keep all* maintains diversity, but loses some accuracy due to staleness. *Sibyl (patching and pruning)* strikes a balance, adding measurements to its corpus over time to generate more candidates while minimizing the impact of staleness by patching paths likely to be out of date. Appendix G evaluates the coverage and accuracy of *Sibyl*'s various approaches to patching and pruning stale paths.

9 Related work

Traceroute tool: Van Jacobson's traceroute tool [26] first enabled measurements of the Internet route from the machine on which the tool is executed to any destination. Followup work addressed limitations in the tool. Paris traceroute modified traceroute to account for load balancing [4]; reverse traceroute enabled a source to measure the route back to it from any destination [30]; and researchers assessed how common interpretations of the tool's output can lead to overestimating route changes [44]. *Sibyl* goes beyond enabling measurement of the route from/to a specific source, and instead chooses (source, destination) pairs that it should measure in order to obtain routes that match specified input criteria.

Measurement platforms and systems: Many distributed platforms have been deployed to cater to the needs of researchers and network operators to perform measurements of Internet routing. DIMES [57], Ark [2], public traceroute servers [63], and RIPE Atlas [55] explicitly serve this goal, whereas other platforms such as PlanetLab [53], MobiPerf [72], and Dasu [56] enable traceroutes among several other capabilities. Leveraging the measurement capabilities offered by these platforms, a large number of systems have been developed that rely on making measurements of the Internet for various purposes such as topology discovery [2], fault diagnosis [31, 32, 74], prefix hijack detection [76], etc. In all of these cases, researchers have relied on issuing traceroutes along paths whose routes match particular criteria relevant to their system, but they have only used small numbers of vantage points due to the overhead of incorporating different platforms and the difficulty in discerning which measurements will be most useful. *Sibyl* can enable these prior systems as well as future ones to take advantage of available measurement platforms.

Studies of Internet routing: Several research efforts have studied the temporal stability of Internet routes [15, 52], attempted to infer routing policies [1, 3, 21, 27, 45], and modeled the evolution of the Internet's topology [49]. We similarly model properties of Internet routing, in our case in service of identifying the measurements that are most beneficial for *Sibyl* in serving user queries.

Route prediction: Many prior efforts have developed techniques to predict Internet routing at the AS [43, 54] and PoP [37, 40, 41] levels. However, in our results, even the state of the art prediction techniques offer only 68% accuracy in correctly predicting AS-level paths. Therefore, instead of attempting to predict a single route for any (source, destination) pair, we focus on estimating the probability that the route will match a query; our approach shows significant gains in prediction accuracy.

10 Conclusion

Internet route measurements are crucial to our ability to troubleshoot and understand the Internet, yet our interface to them remains crude: for decades, the only query that has been easy to answer is, *"What is the path from here to there?"* This limitation leads to inefficient approaches and incomplete understanding. We built and evaluated *Sibyl*, a system that accepts regular expression-based queries and returns fresh path measurements matching the queries. To achieve broad coverage, *Sibyl* includes vantage points (such as traceroute servers and RIPE Atlas probes) that are severely rate-limited, which led to the central challenge in building the system—how can it accurately respond even though, for many queries, it will not have issued traceroutes that match in the recent past? Therefore, we designed *Sibyl* to predict which measurements, if issued, will help fulfill queries, in order to efficiently service requests while subscribing to rate limits. Our evaluation shows that these predictions allow *Sibyl* to easily outpace other schemes in its ability to answer questions about Internet routes, performing nearly as well as if it had access to an oracle to tell it which measurements to issue.

Acknowledgements

We would like to thank our shepherd Monia Ghobadi and the anonymous reviewers for their valuable feedback. The RIPE NCC and Comcast supported our use of RIPE Atlas. Conversations with Shaddin Dughmi and Nate Foster shaped *Sibyl*'s optimization and query language, respectively. This work was supported in part by the National Science Foundation grants CNS-1351100 and CNS-1413978, CNPq, and FAPEMIG.

Appendices

A Optimization details

***Sibyl*'s optimization has good greedy performance.** While the constraints in Eq. 1 (§4.2) enforce multiple budgets, we designed them so that each traceroute only counts against one budget, and so the constraints function as a partition matroid [68]. The utility functions we use for *existence queries* and *diversity queries* exhibit diminishing returns as we add to the set of traceroutes to issue, and so the objective function is submodular (essentially, a set function that displays diminishing returns) [47]. The greedy optimization of submodular functions given partition constraints both has a good theoretical lower bound [16, 68][11] and has been frequently observed to be near-optimal in practice.

In addition to the greedy heuristic only being guaranteed to find a solution within a factor of optimal, the optimization problem itself can lose utility compared to a global optimal due to the following factors:

- Candidate generation can miss useful traceroutes, if no previous traceroutes splice to generate the candidate.
- Prediction errors can lead to errors in expected utility.
- Our formulation assumes the correctness of different predictions is independent, but destination-based routing [19] and other factors mean that the correctness of different predictions may be intertwined.

Section 8.1 assessed the first two factors. The third is an interesting future direction for improving predictions.

Utility functions supported by *Sibyl*. Section 4.2 formalizes the utility functions supported by our UI, but, in general, *Sibyl* will work with any utility function f_q for a query q that satisfies the following properties:

- f_q takes a set of traceroutes T and returns a nonnegative value.
- $f_q(T) > 0$ if and only if $\exists t \in T$ that satisfies q.
- Non-decreasing: $f_q(T) \leq f_q(T \cup \{t\})\ \forall T, \forall t$.
- $S \subseteq T \Rightarrow f_q(S \cup \{t\}) - f_q(S) \geq f_q(T \cup \{t\}) - f_q(T)$ $\forall S, \forall T, \forall t$. In other words, adding additional traceroutes provides diminishing returns.
- The expected utility of issuing a set of traceroutes must be computable within *Sibyl*'s prediction framework, in which a traceroute is predicted as a set of PoP-level paths, each with a confidence.

To be computationally efficient, the expected utility function should also be "incrementally computable": if *Sibyl* already calculated $\mathbb{E}[f_q(T)]$, then computing $\mathbb{E}[f_q(T \cup \{t\})]$ takes time proportional to the time to calculate $\mathbb{E}[f_q(\{t\})]$, not proportional to $|T|$.

B Assessing path stability

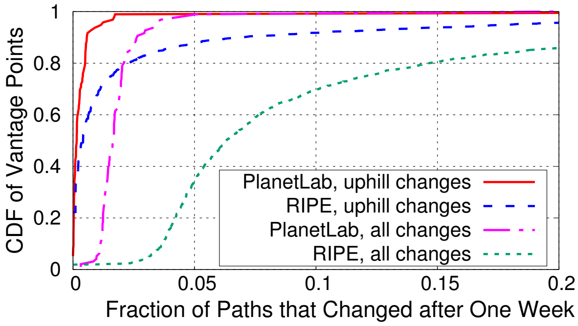


Figure 10: Fraction of paths for which AS-level routes differ in snapshots measured a week apart.

Section 5 describes how *Sibyl* predicts paths by splicing the small number of traceroutes from resource-constrained vantage points onto traceroutes from less-constrained vantage points to a large number of destinations. We assessed path stability to justify this approach. We probed 1000 prefixes from all PlanetLab sites and from 2000 RIPE Atlas vantage points. We repeated these measurements twice, a week apart. Figure 10 shows, for every vantage point, the fraction of prefixes for which the AS-level routes differ across a week, revealing more RIPE Atlas paths change than PlanetLab paths.

Internet paths are generally considered to have an uphill portion, traversing from customers to providers, followed by a downhill portion from providers to customers, possibly with a peering link in between. Figure 10 also plots the fraction of prefixes that have different AS-level routes in the two snapshots if we consider only the uphill portions of the paths. The uphill paths differ much less frequently than the full paths, implying that most of the differences are on the downhill portions of paths. By combining the uphill (more stable) portion of paths from

[11] The greedy algorithm we use has an approximation ratio of 0.5. A randomized variant has a ratio of $1 - 1/e \approx 0.63$ [68].

rate-limited RIPE Atlas/traceroute server vantage points with the downhill portions of (frequently-refreshed) PlanetLab paths, *Sibyl* minimizes the impact of path instability on its predictions.

C Features used by RuleFit

In this section we provide more details on the RuleFit model we train to estimate the correctness of a path prediction (§5.2). To identify important features, we adopted a multi-round refinement process, starting from a large set of features that we reduced each round, retaining features RuleFit found to have predictive power. We describe features retained at the end of this process.

Source path features: The greatest challenge in identifying which spliced path is correct is to pick the correct route out from the source, since Internet routing is predominantly destination-based and the source's portion of the source path has a destination different from the one we are predicting. Traffic engineering practices such as hot and cold potato routing may also exacerbate this issue. We characterize the source path using the following features: the number of PoPs and ASes along the source path, the round-trip latency from the source to the splice point [39], and the degree of the source AS (from CAIDA data [3]). The intuition behind these features is that a prediction is more likely to be correct if the source's part of the path is short (so quickly intersects a known path to the destination) and if the source AS is small (so has fewer routing options we can incorrectly pick). We do not consider the age of measurements as a feature since we take steps to prune out-of-date measurements, as described in Section 7. Most (source, destination) pairs have a path that is prevalent over long time periods [15, 52].

Splice point features: We considered the characteristics of the splice point as additional features such as (i) the type of AS splice point (i.e. educational, transit, access, transit/access, content, enterprise, educational/research, non-profit network, from CAIDA data [10]); and (ii) the business relationship between the AS of the splice point and its neighbors in the predicted path (from CAIDA data [38]). The type and the AS relationships allow RuleFit to learn to favor spliced paths that follow common routing policies, such as valley-free [21].

iPlane-derived features: iPlane picks the correct spliced path more often than not [39], and so, for each spliced path, we calculate the features that iPlane uses, as well as comparisons between that spliced path and the one that iPlane picks (inflation in terms of RTT up to the splice point and in terms of AS- and PoP-level path lengths). We also included the rank order assigned by iPlane to the spliced path, to account for mechanisms added to improve iPlane's prediction accuracy [41].

Spliced Path Feature	Importance
1. PoP-level similarity with the other paths	1
2. PoP-level path length inflation vs iPlane's top-ranked path	.90
3. Total number of PoP splice points	.60
4. Total number of AS splice points	.55
5. AS splice point type	.52
6. AS splice point relationship with neighbors	.49
7. Number of PoPs in iPlane's top-ranked path	.44
Other features	≤ .34

Table 1: Feature importance according to RuleFit.

Spliced path set features: Finally, we compute some features by comparing the spliced path with the other spliced paths from the vantage point to the destination. We used the Jaccard Index to estimate the average similarity between the spliced path and other paths both at the PoP and AS level. We aim to inform RuleFit whether or not the other paths confirm this one. We also include as features the total number of spliced paths and the total number of ASes containing splice points.

Most important features: RuleFit computes the importance of each rule as a function of how often it gets applied and how much it impacts the correctness of the prediction. For each feature, it computes this as the sum of the importance of the rules that use the feature.

Table 1 reports the resulting ordering of features with normalized importance computed by RuleFit. Several features turned out to play an important role in estimating the similarity of a spliced path to the true path. The first, third, and fourth most important features capture how similar the spliced paths are; intuitively, if there are few splicing points and all spliced paths are similar, then there is less diversity and spliced paths are likely similar to the true path. The second and seventh most important feature follows from Internet routing protocols that prefer short paths. The fifth and sixth most important features capture AS routing relationships at the splicing point, which enables RuleFit, e.g., to reduce confidence in splices that violate the valley-free model.

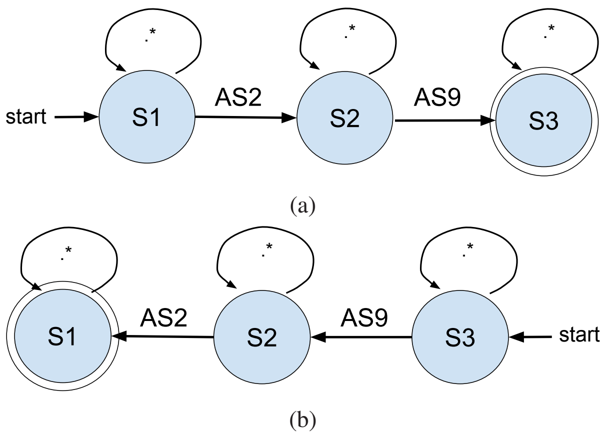
D Examples

D.1 Candidate generation

We first provide an example of how *Sibyl* generates candidate traceroutes to consider issuing (§6). For ease of exposition, assume that an IP address maps to an AS and PoP corresponding to the address's first octet (e.g., 1.0.0.1 is in AS1 and PoP1; 5.0.0.1 is in AS5, PoP5). Assume *Sibyl* has three existing traceroutes it can combine to generate new candidates:

1. 1.0.0.1 (AS1, PoP1), 2.0.0.1 (AS2, PoP2), 3.0.0.1 (AS3, PoP3), 4.0.0.1 (AS4, PoP4), 5.0.0.1 (AS5, PoP5)

2. 6.0.0.1 (AS6, PoP6), 7.0.0.1 (AS7, PoP7), 8.0.0.1 (AS8, PoP8), 9.0.0.1 (AS9, PoP9), 10.0.0.1 (AS10, PoP10)



(a)

(b)

Figure 11: (a) Forward and (b) reverse FSAs corresponding to a query for a traceroute through AS2 and AS9.

3. 11.0.0.1 (AS11, PoP11), 12.0.0.1 (AS12, PoP12), 3.0.0.1 (AS3, PoP3), 9.0.0.1 (AS9, PoP9), 13.0.0.1 (AS13, PoP13)

Say a user issues an existence query: "I want a traceroute that traverses AS2 and AS9, in that order, consecutively or not," is expressed as the following regular expression:

```
^.*AS2.*AS9.*$.
```

This regular expression is then translated to an FSA shown in Figure 11(a). *Sibyl* then runs the FSA over each traceroute, maintaining a record of the transitions in the FSA taken when consuming the PoPs in each of its existing traceroutes, as shown in Table 2(a).

Next, the FSA is reversed (Figure 11(b)), and the reverse FSA is run over the traceroutes from destination to source. Table 2(b) shows the transitions used in this case.

In our example, PoP 3 is labeled with the transition ($S_2 \xrightarrow{.*} S_2$) when the forward FSA is applied on Trace 1, and the same PoP is labeled with the reverse of that transition when the reverse FSA is applied on Trace 3. Hence, *Sibyl* splices Traceroute 1 (PoP1→PoP2→PoP3. . .) and Traceroute 3 (. . . PoP3→PoP9→PoP13) at PoP3 to generate a candidate. The candidate pair is constructed from the source of Traceroute 1 and the destination of Traceroute 3 which gives (1.0.0.1, 13.0.0.1).

D.2 Likelihood estimation

We next walk through an example of how *Sibyl* calculates how likely a traceroute is to satisfy a query (Eq. 4 in §5). Assume that, in addition to (1.0.0.1, 13.0.0.1), *Sibyl* also finds (15.0.0.1, 16.0.0.1) as a possible candidate. Once *Sibyl* identifies the candidates for a query, it uses iPlane to generate a set of possible paths for each candidate (source, destination) pair. *Sibyl* uses its RuleFit-trained model to estimate the Jaccard indexes for each spliced path compared to the corresponding (unknown) actual path. It uses these estimates to compute the likelihood of each candidate matching the query. Consider the example paths and estimated Jaccard indexes in Table 3, where we show AS-level paths for ease of exposition.

Forward FSA Transition	**PoPs Traversed**
$S_1 \xrightarrow{.*} S_1$	Trace 1: PoP 1, 2, 3, 4, 5 Trace 2: PoP 6, 7, 8, 9, 10 Trace 3: PoP 11, 12, 3, 9, 13
$S_1 \xrightarrow{\text{AS2}} S_2$	Trace 1: PoP 2
$S_2 \xrightarrow{.*} S_2$	Trace 1: PoP 3, 4, 5
$S_2 \xrightarrow{\text{AS9}} S_3$	-
$S_3 \xrightarrow{.*} S_3$	-

(a)

Reverse FSA Transition	**PoPs Traversed**
$S_3 \xleftarrow{.*} S_3$	Trace 1: PoP 5, 4, 3, 2, 1 Trace 2: PoP 10, 9, 8, 7, 6 Trace 3: PoP 13, 9, 3, 12, 11
$S_2 \xleftarrow{\text{AS9}} S_3$	Trace 2: PoP 9 Trace 3: PoP 9
$S_2 \xleftarrow{.*} S_2$	Trace 2: PoP 8, 7, 6 Trace 3: PoP 3, 12, 11
$S_1 \xleftarrow{\text{AS2}} S_2$	-
$S_1 \xleftarrow{.*} S_1$	-

(b)

Table 2: Transitions activated by PoPs in each traceroute on the (a) forward and (b) reverse FSAs.

Candidate	Splice	Jaccard	Predicted AS-Level Path
(1.0.0.1, 13.0.0.1)	A	0.7	AS1 AS2 AS3 AS9 AS13
	B	0.5	AS1 AS20 AS21 AS9 AS13
(15.0.0.1, 16.0.0.1)	C	0.6	AS15 AS2 AS3 AS9 AS16
	D	0.6	AS15 AS2 AS4 AS9 AS16

Table 3: All spliced paths for each candidate and their RuleFit-predicted Jaccard indexes.

For the candidate pair (1.0.0.1, 13.0.0.1), *Sibyl* estimated that spliced path A is more likely to be correct than spliced path B (0.7 vs 0.5), which (via §5.2) normalize to $0.41 = 0.7 \times 0.7/(0.7+0.5)$ and $0.29 = 0.7 \times 0.5/(0.7+0.5)$. Spliced path A matches the user's query, whereas B does not traverse AS2. The final likelihood that candidate (1.0.0.1, 13.0.0.1) matches the query is 0.41, from Eq. 4.

For (15.0.0.1, 20.0.0.1), spliced paths C and D have lower estimated Jaccard indexes than spliced path A, but both satisfy the user's query. These spliced paths result in a likelihood of matching the query equal to $0.6 = 0.6 \times 0.6/(0.6+0.6) + 0.6 \times 0.6/(0.6+0.6)$, making (15.0.0.1, 16.0.0.1) a stronger candidate to satisfy the user's query.

D.3 Diversity queries

To illustrate the usefulness of diversity queries, we will use an example of *Sibyl* finding diverse AS paths that a system such as iSpy [76] can use to monitor for prefix hijacks in BGP. Since issuing traceroutes from all vantage points is not feasible, we want *Sibyl* to find a set of vantage points to use that maximizes AS path coverage to a prefix 204.57.0.0/21. Since we want to diversify over AS, we build the following query:

```
^{.*}-204.57.0.0/21$ by AS
```

For simplicity of exposition, we assume *Sibyl* predicts a single path for each candidate and has complete confidence in all predictions, removing the probabilistic expected value calculation of Eq. 3. *Sibyl* predicts traceroutes with the following AS paths toward 204.57.0.0/21:

1. AS3356, AS209, AS2722, AS47
2. AS1299, AS10490, AS2722, AS47
3. AS3257, AS209, AS2722, AS47
4. AS1273, AS209, AS2722, AS47
5. AS6939, AS226, AS2914, AS2497, AS47
6. AS3257, AS209, AS2722, AS47
7. AS701, AS2914, AS209, AS2722, AS47

Sibyl greedily selects traceroutes that offer the highest diversity utility first. The greedy selection starts out with an empty AS set. Traceroutes are then selected based on how many new ASes a path is predicted to add. In the above example, Traceroute 5 is selected first since it has a utility of 5 ASes (contains 5 new ASes). Traceroute 7 would be greedily selected next since it has a marginal utility of 4 ASes. The current AS set is now:

```
AS6939, AS226, AS2914, AS2497, AS47, AS701,
AS2914, AS209, AS2722
```

Of the remaining traceroutes, Traceroutes 1, 3, 4, and 6 each offer only one new AS compared to the above set, whereas Traceroute 2 has 2 new ASes. Hence, Traceroute 2 is selected. In subsequent rounds, Traceroutes 3 and 4 would be selected if budget allowed, but Traceroute 6 would not be since it adds no new ASes.

E Evaluation of RuleFit model

Section 8.3 showed that *Sibyl*'s estimates of how likely a candidate traceroute is to satisfy a query are accurate enough to use as expected utilities. In this section we look at the distribution of likelihood values across candidates and at the accuracy of the Jaccard index estimates (§5.2) that *Sibyl* uses to calculate the likelihoods.

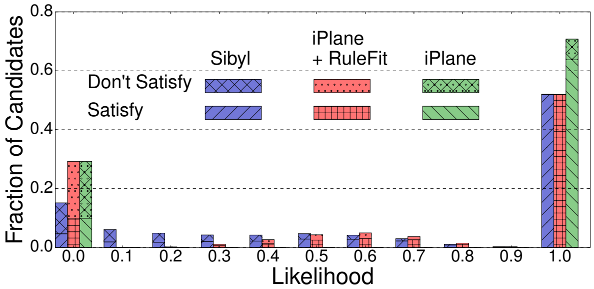


Figure 12: Distribution of candidates by likelihood.

Distribution of likelihood estimates. Figure 12 partitions the range of likelihood values ([0, 1]) into 11 buckets ([0, 0.05], [0.05, 0.15], ..., [0.95, 1]), and shows the number of candidates in each bucket, broken down by whether the candidates satisfy their queries or not. We also add two comparison points: (1) iPlane: iPlane provides a single predicted path for a candidate and does not have a notion of varying confidence [40], and so we assign a candidate a likelihood of 1 if iPlane's prediction matches the query and 0 if it does not. (2) iPlane with confidence ranking: for iPlane predictions that match their queries, we extend iPlane by assigning a likelihood equal to our RuleFit model's estimated confidence in the prediction. As seen in the graph, *Sibyl*'s likelihood estimation provides benefit over iPlane. In the bucket of likelihood [0.95, 1], *Sibyl* only includes candidates that satisfy queries, while iPlane includes some candidates that do not satisfy queries. *Sibyl* only assigns a likelihood of 1 to a candidate when all its spliced paths satisfy the query *and* RuleFit rates it high confidence. *Sibyl* also provides benefit over iPlane by removing some candidates that can satisfy queries from the [0, 0.05] bucket. This improvement comes at the cost of moving some candidates that do not satisfy queries from the [0, 0.05] likelihood bucket to other low-likelihood buckets, which we consider to be acceptable since *Sibyl* gives low priority to issue measurements for candidates with low likelihood.

Together, Figures 8 and 12 show that *Sibyl* computes likelihoods that can reasonably reflect the probability of matching a query, and it assigns most candidates either very high or very low likelihood values, enabling it to distinguish between candidates that it should or should not select to satisfy queries.

Accuracy of confidence values. Figure 13 evaluates RuleFit's capability to predict the PoP-level similarity of spliced paths to the actual paths they are predicting, which it does without access to the actual paths. We use RuleFit to estimate the Jaccard index for 4 million spliced paths (not included in the training set), then calculate the actual Jaccard index by comparing the spliced path to

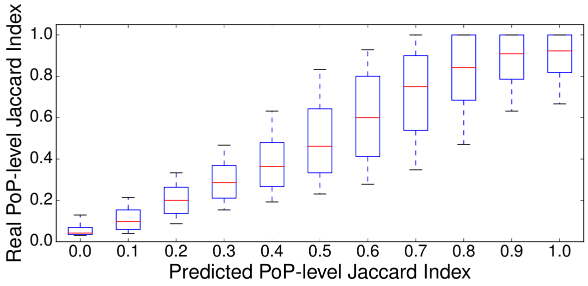


Figure 13: Spliced paths' predicted vs actual Jaccard index with respect to the actual path.

the actual path. We group the spliced paths by their predicted Jaccard index and show the 10th, 25th, 50th, 75th, and 90th percentiles of the true Jaccard values for each group. We see that our estimated Jaccard indexes are well correlated to the true Jaccard values.

F Queries used in evaluation

We used several types of queries when evaluating *Sibyl* (§8.1). For each traceroute that *Sibyl* does not have access to, we generate all possible queries that the traceroute matches for the following query types:

1. `^.*A.*D$` Traverse A on the way to destination D.
2. `^[^A]+A.*D$` Traverse, but do not start at, A on the way to destination D.
3. `^.*AB.*D$` Traverse link A-B on the way to destination D.
4. `^.*A.*B.*C.*$` Traverse A, B, and C in sequence.

Among all possible queries of these types, our evaluation randomly selected an equal number of each type. Queries of types 1 and 2 represent queries reverse traceroute uses as part of its measurements [30]. Query type 3 represents queries operators might ask when troubleshooting performance problems towards a destination, to assess paths that use a particular link. All three look for routes toward a destination *D* traversing a specific network region. Query type 4 does not specify a destination and could be used to study inter-AS routing policing and business relationships [38] or to look for routes that take long detours in between two nearby hops (e.g., [22]).

G Efficacy of staleness patching & pruning

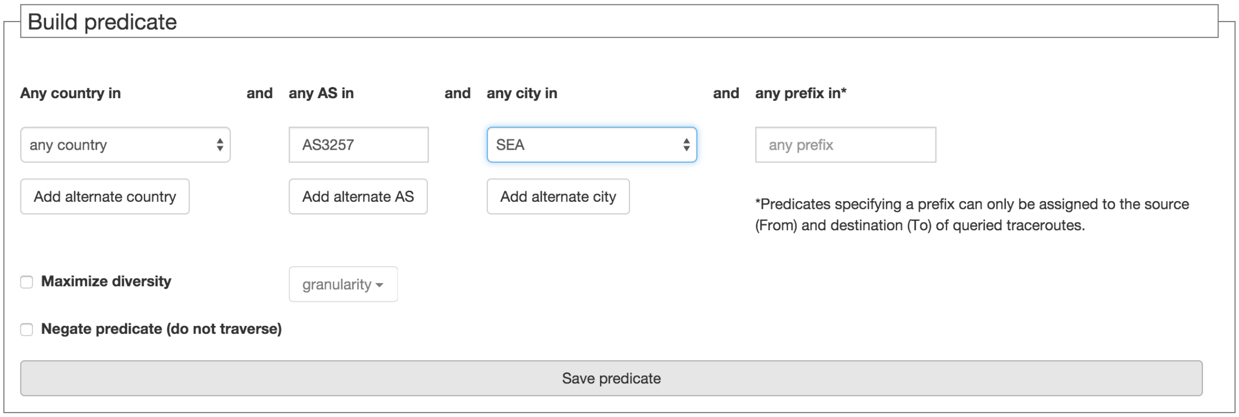
Section 8.4 evaluated the impact of staleness on *Sibyl*'s end-to-end ability to satisfy queries, showing that its techniques for dealing with stale measurements allow it to outperform techniques that either keep or discard all old measurements. In this section we evaluate the accuracy and coverage of its techniques (§7) in isolation.

Traceroute-based source/destination patching. First, we validate *Sibyl*'s approaches of using a path change observed on one path to update other previously measured paths (from the same source or to the same destination) that traverse the path segment that changed. For this, we issued traceroutes from all PlanetLab sites to 150K prefixes on Dec. 5 and on Dec. 6 2014. We calculated the probability that paths undergo identical path changes, given their Dec. 5 routes traversed a shared segment that changed in one of the routes on Dec. 6. For 65% of path changes, all paths experience an identical change. Results on measurements one week apart are similar.

BGP-based destination pruning. We evaluate *Sibyl*'s BGP-based filtering of stale paths on RIPE Atlas measurements gathered between July 2 and August 27, 2015, using daily BGP paths from BGPStream [50]. We mapped the traceroute destinations to the longest prefix in the collected BGP data, excluding prefixes longer than 24.

First, for coverage, of the (AS, destination) pairs in our traceroutes, only 5% of the ASes appear in BGP feed paths towards the destinations, demonstrating both the superior coverage of our traceroute vantage points compared to available BGP feeds and also a limitation with BGP-based filtering. However, 84% of our traceroutes include at least one pair seen in the BGP feeds. Of the pairs seen in both data sources, the AS paths are the same in 57% of cases. The other 43% reflect a mix of large ASes using multiple paths, of errors in translating traceroutes to AS paths, and of misalignment in time because we do not have an exact timestamp for the traceroutes.

Second, we evaluate the accuracy of BGP-based filtering. Every time we refreshed an Atlas traceroute to a destination *d*, for every AS *A* on the traceroute, we check three conditions. 1:(*BGP-change*) Is the BGP path to *d* different than it was at the time of the original traceroute to *d*? 2:(*TR-change*) Did *A*'s traceroute AS path change between the two measurements? 3:(*TR-match*) Did *A*'s original traceroute AS path match *A*'s BGP path at the time it was issued? Comparing every instance of *BGP-change* with the subset that are also *TR-change*, 72% of BGP changes were also reflected in traceroutes. Comparing instances that are both *BGP-change* and *TR-change* with the subset that are also *TR-match*, the percentage increases to 77% if we add the stricter condition that the BGP and traceroute paths matched to begin with. Overall, BGP monitoring prunes 9% of the traceroute changes if we require the *TR-match* check and 13.8% if we do not.



Build predicate

Any country in: any country

and any AS in: AS3257

and any city in: SEA

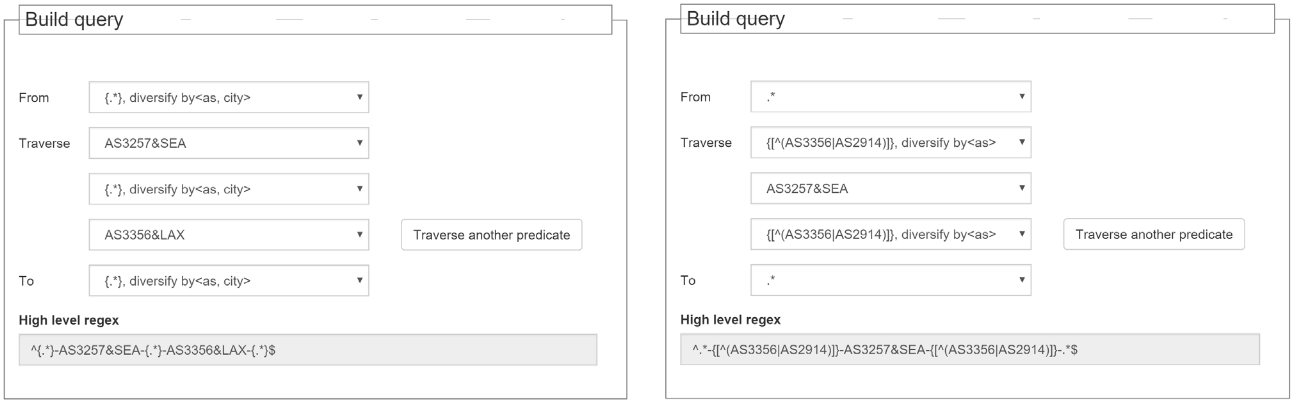
and any prefix in*: any prefix

*Predicates specifying a prefix can only be assigned to the source (From) and destination (To) of queried traceroutes.

□ Maximize diversity

□ Negate predicate (do not traverse)

Figure 14: Screenshot of *Sibyl*'s interface to build predicates.



Build query

From: {.*}, diversify by<as, city>

Traverse: AS3257&SEA

{.*}, diversify by<as, city>

AS3356&LAX

To: {.*}, diversify by<as, city>

High level regex: ^{.*}-AS3257&SEA-{.*}-AS3356&LAX-{.*}$

Build query

From: .*

Traverse: {[^(AS3356|AS2914)]}, diversify by<as>

AS3257&SEA

{[^(AS3356|AS2914)]}, diversify by<as>

To: .*

High level regex: ^.*-{[^(AS3356|AS2914)]}-AS3257&SEA-{[^(AS3356|AS2914)]}-.*$

(a) All paths traversing GTT in Seattle then Level3 in Los Angeles. (b) Paths through GTT peers other than NTT and Level3 in Seattle.

Figure 15: Screenshots of example queries from Section 3.2, built by composing predicates such as the one in Fig. 14.

H Unsatisfiable Queries

Section 8 uses queries that are satisfiable–since we generate them from traceroutes *Sibyl* could choose to issue. Here we evaluate whether *Sibyl* can avoid wasting budget on queries it has no hope of satisfying, to avoid having them impede its performance on queries it can satisfy. We generated sensible unsatisfiable queries by generating existence queries as in Section 8.1, removing *Sibyl*'s access to 10% of the RIPE Atlas and traceroute server VPs, then identifying queries that can only be satisfied by measurements from the removed vantage points.[12]

In our experiment, we add unsatisfiable queries to the set of queries submitted to *Sibyl* while keeping the probing budget fixed. As we move from all queries satisfiable to an even mix of satisfiable and unsatisfiable, *Sibyl* still matches just as many queries, 76% on average as in Figure 3. It does generate some candidates to consider issuing for some of the unsatisfiable queries. However, *Sibyl*'s ability to rate the likelihood of matching allows it to prioritize measurements with high expected utility, concentrating budget on queries that can be satisfied. In practice, it could inform a user when it had no candidates likely to match the user's query.

To verify that this result was because the system assessed that its vantage points were unable to satisfy the queries, not because it found the queries to be unsatisfiable in general, we reintroduced the 10% of vantage points back into the system and ran it with just the previously unsatisfiable queries. *Sibyl* satisfied an average of 48% of the queries, suggesting that they are hard but not impossible when suitable vantage points are available. When we then combined the two batches of queries, increasing the absolute traceroute budget to maintain the 1:1 query:budget ratio, *Sibyl* satisfied an average of 58% of queries, balancing the budget well across the two sets to nearly equal the $(76+48)/2 = 62\%$ average performance when it could dedicate itself to one set.

I *Sibyl*'s query interface

We built a web-based user interface to guide users in specifying queries. Figure 14 presents a screenshot of the widgets used to build a predicate. Users can build

[12] We do not include trivial unsatisfiable queries such as asking for paths originated from ASes hosting the removed VPs.

a broad class of predicates that accept (or, via negation, reject) a user-specified set of values (e.g., particular cities or ASes) at any or all granularities. Users can then build queries by specifying a sequence of predicates they want paths to traverse. Figures 15(a) and (b) show examples of simplified versions of queries from Section 3.2.

References

[1] Anwar, R., Niaz, H., Choffnes, D., Cunha, I., Gill, P., and Katz-Bassett, E. Investigating interdomain routing policies in the wild. In *IMC* (2015).

[2] Archipelago measurement infrastructure. `http://www.caida.org/projects/ark/`.

[3] AS Rank: AS Ranking. `http://as-rank.caida.org/`.

[4] Augustin, B., Cuvellier, X., Orgogozo, B., Viger, F., Friedman, T., Latapy, M., Magnien, C., and Teixeira, R. Avoiding traceroute anomalies with Paris traceroute. In *IMC* (2006).

[5] Augustin, B., Friedman, T., and Teixeira, R. Measuring multipath routing in the Internet. *IEEE/ACM TON* (2011).

[6] Augustin, B., Krishnamurthy, B., and Willinge, W. IXPs: mapped? In *IMC* (2009).

[7] Banerjee, R., Chiang, L., Mishra, A., Razaghpanah, A., Sekar, V., Choi, Y., and Gill, P. Internet outages, the eyewitness accounts: Analysis of the outages mailing list. In *PAM* (2015).

[8] Bourgeau, T., Augé, J., and Friedman, T. TopHat: supporting experiments through measurement infrastructure federation. In *TridentCom* (2010).

[9] Bu, T., Duffield, N., Presti, F. L., and Towsley, D. Network tomography on general topologies. In *SIGMETRICS* (2002).

[10] CAIDA UCSD AS classification dataset. `http://www.caida.org/data/as_classification.xml`.

[11] Chang, D., Govindan, R., and Heidemann, J. The temporal and topological characteristics of BGP path changes. In *ICNP* (2003).

[12] Chen, K., Choffnes, D. R., Potharaju, R., Chen, Y., Bustamante, F. E., Pei, D., and Zhao, Y. Where the sidewalk ends: Extending the Internet AS graph using traceroutes from P2P users. In *CoNEXT* (2009).

[13] Chiu, Y.-C., Schlinker, B., Radhakrishnan, A. B., Katz-Bassett, E., and Govindan, R. Are we one hop away from a better Internet? In *IMC* (2015).

[14] Coates, M., Hero, A., Nowak, R., and Yu, B. Internet tomography. *IEEE Signal Processing Magazine* (2002).

[15] Cunha, I., Teixeira, R., Veitch, D., and Diot, C. Predicting and tracking Internet path changes. In *SIGCOMM* (2011).

[16] Dughmi, S. Submodular functions: Extensions, distributions, and algorithms. A survey (PhD qualifying exam report). Tech. rep., Stanford, 2009.

[17] Fanou, R., Francois, P., and Aben, E. On the diversity of interdomain routing in Africa. In *PAM* (2015).

[18] Flach, T., Katz-Bassett, E., and Govindan, R. Quantifying violations of destination-based forwarding on the Internet. In *IMC* (2012).

[19] Flach, T., Katz-Bassett, E., and Govindan, R. Quantifying violations of destination-based forwarding on the Internet. In *IMC* (November 2012).

[20] Friedman, J. H., and Popescu, B. E. Predictive learning via rule ensembles. *The Annals of Applied Statistics* (2008), 916–954.

[21] Gao, L., and Rexford, J. Stable Internet routing without global coordination. In *SIGMETRICS* (2000).

[22] Gupta, A., Calder, M., Feamster, N., Chetty, M., Calandro, E., and Katz-Bassett, E. Peering at the Internet's frontier: A first look at ISP interconnectivity in Africa. In *PAM* (2014).

[23] Hengartner, U., Moon, S., Mortier, R., and Diot, C. Detection and analysis of routing loops in packet traces. In *IMW* (2002).

[24] Hiran, R., Carlsson, N., and Gill, P. Characterizing large-scale routing anomalies: a case study of the China Telecom incident. In *PAM* (2013).

[25] Internet address hitlist dataset, PREDICT ID USC LANDER internet_address_hitlist_it28wbeta20090914. `http://www.isi.edu/ant/lander`.

[26] Jacobson, V. Traceroute. `ftp://ftp.ee.lbl.gov/traceroute.tar.gz`.

[27] Javed, U., Cunha, I., Choffnes, D. R., Katz-Bassett, E., Anderson, T., and Krishnamurthy, A. PoiRoot: Investigating the root cause of interdomain path changes. In *SIGCOMM* (2013).

[28] John, J. P., Katz-Bassett, E., Krishnamurthy, A., Anderson, T., and Venkataramani, A. Consensus routing : The Internet as a distributed system. In *NSDI* (2008).

[29] Katz-Bassett, E., John, J. P., Krishnamurthy, A., Wetherall, D., Anderson, T., and Chawathe, Y. Towards IP geolocation using delay and topology measurements. In *IMC* (2006).

[30] Katz-Bassett, E., Madhyastha, H. V., Adhikari, V., Scott, C., Sherry, J., van Wessep, P., Anderson, T., and Krishnamurthy, A. Reverse traceroute. In *NSDI* (2010).

[31] Katz-Bassett, E., Madhyastha, H. V., John, J. P., Krishnamurthy, A., Wetherall, D., and Anderson, T. Studying black holes in the Internet with Hubble. In *NSDI* (2008).

[32] Katz-Bassett, E., Scott, C., Choffnes, D. R., Cunha, I., Valancius, V., Feamster, N., Madhyastha, H. V., Anderson, T. E., and Krishnamurthy, A. LIFEGUARD: Practical repair of persistent route failures. In *SIGCOMM* (2012).

[33] Keys, K., Hyun, Y., Luckie, M., and kc claffy. Internet-scale IPv4 alias resolution with MIDAR: System architecture. Tech. rep., Cooperative Association for Internet Data Analysis (CAIDA), 2011.

[34] Krishnan, R., Madhyastha, H. V., Srinivasan, S., Jain, S., Krishnamurthy, A., Anderson, T., and Gao, J. Moving beyond end-to-end path information to optimize CDN performance. In *IMC* (2009).

[35] Kushman, N., Kandula, S., and Katabi, D. Can you hear me now?! It must be BGP. *SIGCOMM CCR* (2007).

[36] Labovitz, C., Ahuja, A., Bose, A., and Jahanian, F. Delayed Internet routing convergence. In *SIGCOMM* (2000).

[37] Lee, D., Jang, K., Lee, C., Iannaccone, G., and Moon, S. Scalable and systematic Internet-wide path and delay estimation from existing measurements. *Computer Networks* (2011).

[38] Luckie, M., Huffaker, B., Dhamdhere, A., Giotsas, V., and Claffy, K. AS relationships, customer cones, and validation. In *IMC* (2013).

[39] Madhyastha, H. V., Anderson, T., Krishnamurthy, A., Spring, N., and Venkataramani, A. A structural approach to latency prediction. In *IMC* (2006).

[40] Madhyastha, H. V., Isdal, T., Piatek, M., Dixon, C., Anderson, T., Krishnamurthy, A., and Venkataramani, A. iPlane: An information plane for distributed services. In *OSDI* (2006).

[41] Madhyastha, H. V., Katz-Bassett, E., Anderson, T., Krishnamurthy, A., and Venkataramani, A. iPlane Nano: Path prediction for peer-to-peer applications. In *NSDI* (2009).

[42] Mahajan, R., Zhang, M., Poole, L., and Pai, V. Uncovering performance differences among backbone ISPs with Netdiff. In *NSDI* (2008).

[43] Mao, Z. M., Qiu, L., Wang, J., and Zhang, Y. On AS-level path inference. In *SIGMETRICS* (2005).

[44] Marchetta, P., Persico, V., Katz-Bassett, E., and Pescape, A. Don't trust traceroute (completely). In *CoNEXT Student Workshop* (2013).

[45] Mühlbauer, W., Uhlig, S., Fu, B., Meulle, M., and Maennel, O. In search for an appropriate granularity to model routing policies. In *SIGCOMM* (2007).

[46] Narayana, S., Tahmasbi, M., Rexford, J., and Walker, D. Compiling path queries. In *NSDI* (2016).

[47] Nemhauser, G. L., Wolsey, L. A., and Fisher, M. L. An analysis of approximations for maximizing submodular set functions I. *Mathematical Programming 14* (1978), 265–294.

[48] Oliveira, R., Pei, D., Willinger, W., Zhang, B., and Zhang, L. In search of the elusive ground truth: The Internet's AS-level connectivity structure. In *SIGMETRICS* (2008).

[49] Oliveira, R. V., Zhang, B., and Zhang, L. Observing the evolution of Internet AS topology. In *SIGCOMM* (2007).

[50] Orsini, C., King, A., and Dainotti, A. BGPStream: a software framework for live and historical BGP data analysis. Tech. rep., Center for Applied Internet Data Analysis (CAIDA), Oct 2015.

[51] Outages mailing list. `http://isotf.org/mailman/listinfo/outages`.

[52] Paxson, V. End-to-end routing behavior in the Internet. *IEEE/ACM TON* (1997).

[53] PlanetLab website. `http://www.planet-lab.org`.

[54] Qiu, J., and Gao, L. AS path inference by exploiting known AS paths. In *GLOBECOM* (2006).

[55] RIPE Atlas. `https://atlas.ripe.net/`.

[56] Sánchez, M. A., Otto, J. S., Bischof, Z. S., Choffnes, D. R., Bustamante, F. E., Krishnamurthy, B., and Willinger, W. A measurement experimentation platform at the Internet's edge. *IEEE/ACM TON* (2014).

[57] Shavitt, Y., and Shir, E. DIMES: Let the Internet measure itself. *SIGCOMM CCR* (2005).

[58] Spring, N., Mahajan, R., and Anderson, T. Quantifying the causes of path inflation. In *SIGCOMM* (2003).

[59] Spring, N., Mahajan, R., and Wetherall, D. Measuring ISP topologies with Rocketfuel. In *SIGCOMM* (2002).

[60] Sridharan, A., Moon, S. B., and Diot, C. On the correlation between route dynamics and routing loops. In *IMC* (2003).

[61] Steenbergen, R. A. A practical guide to (correctly) troubleshooting with traceroute. In *NANOG 45* (2009). `http://www.nanog.org/meetings/nanog45/presentations/Sunday/RAS_traceroute_N45.pdf`.

[62] Sundaresan, S., de Donato, W., Feamster, N., Teixeira, R., Crawford, S., and Pescape, A. Broadband Internet performance: A view from the gateway. In *SIGCOMM* (2011).

[63] Traceroute.org. `http://www.traceroute.org/`.

[64] Trammell, B., Casas, P., Rossi, D., Bar, A., Houidi, Z., Leontiadis, I., Szemethy, T., and Mellia, M. mPlane: an intelligent measurement plane for the Internet. *Communications Magazine, IEEE 52*, 5 (2014), 148–156.

[65] Veanes, M. Applications of symbolic finite automata. In *Implementation and Application of Automata*. Springer, 2013.

[66] Veitch, D., Augustin, B., Teixeira, R., and Friedman, T. Failure control in multipath route tracing. In *INFOCOM* (2009).

[67] Vissicchio, S., Tilmans, O., Vanbever, L., and Rexford, J. Central control over distributed routing. In *SIGCOMM* (2015).

[68] Vondrak, J. Optimal approximation for the submodular welfare problem in the value oracle model. In *STOC* (2008).

[69] Wang, F., Mao, Z. M., Wang, J., Gao, L., and Bush, R. A measurement study on the impact of routing events on end-to-end Internet path performance. In *SIGCOMM* (2006).

[70] Wang, Z., Qian, Z., Xu, Q., Mao, Z., and Zhang, M. An untold story of middleboxes in cellular networks. *SIGCOMM CCR* (2011).

[71] Wong, B., Stoyanov, I., and Sirer, E. G. Octant: A comprehensive framework for the geolocalization of Internet hosts. In *NSDI* (2007).

[72] Xu, Q., Huang, J., Wang, Z., Qian, F., Gerber, A., and Mao, Z. M. Cellular data network infrastructure characterization and implication on mobile content placement. In *SIGMETRICS* (2011).

[73] Zarifis, K., Flach, T., Nori, S., Choffnes, D., Govindan, R., Katz-Bassett, E., Mao, Z. M., and Welsh, M. Diagnosing path inflation of mobile client traffic. In *PAM* (2014).

[74] Zhang, M., Zhang, C., Pai, V., Peterson, L., and Wang, R. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *OSDI* (2004).

[75] Zhang, Y., Mao, Z. M., and Zhang, M. Effective diagnosis of routing disruptions from end systems. In *NSDI* (2008).

[76] Zhang, Z., Zhang, Y., Hu, Y. C., Mao, Z. M., and Bush, R. iSpy: detecting IP prefix hijacking on my own. In *SIGCOMM* (2008).

VAST: A Unified Platform for Interactive Network Forensics

Matthias Vallentin
vallentin@icir.org
UC Berkeley

Vern Paxson
vern@icir.org
UC Berkeley / ICSI

Robin Sommer
robin@icir.org
ICSI / LBNL

Abstract

Network forensics and incident response play a vital role in site operations, but for large networks can pose daunting difficulties to cope with the ever-growing volume of activity and resulting logs. On the one hand, logging sources can generate tens of thousands of events per second, which a system supporting comprehensive forensics must somehow continually ingest. On the other hand, operators greatly benefit from *interactive* exploration of disparate types of activity when analyzing an incident.

In this paper, we present the design, implementation, and evaluation of VAST (Visibility Across Space and Time), a distributed platform for high-performance network forensics and incident response that provides both continuous ingestion of voluminous event streams and interactive query performance. VAST leverages a native implementation of the actor model to scale both intra-machine across available CPU cores, and inter-machine over a cluster of commodity systems.

1 Introduction

Security incidents often leave network operators scrambling to ferret out answers to key questions: How did the attackers get in? What did they do once inside? Where did they come from? What activity patterns serve as *indicators* reflecting their presence? How do we prevent this attack in the future?

Operators can only answer such questions by drawing upon high-quality logs of past activity recorded over extended time. Incident analysis often starts with a narrow piece of intelligence, typically a local system exhibiting questionable behavior, or a report from another site describing an attack they detected. The analyst then tries to locate the described behavior by examining logs of past activity, often cross-correlating information of different types to build up additional context. Frequently, this process in turn produces new leads to explore iteratively ("peeling the onion"), continuing and expanding until ultimately the analyst converges on as complete of an understanding of the incident as they can extract from the available information.

This process, however, remains manual and time-consuming, as no single storage system efficiently integrates the disparate sources of data (e.g., NIDS, firewalls, NetFlow data, service logs, packet traces) that investigations often involve. While standard SIEM systems such as Splunk aggregate logs from different sources into a single database, their data models omit crucial semantics, and they struggle to scale to the data rates that large-scale environments require.

Based on these needs, and drawing upon our years of experience working closely with operational security staff, we formulate three key goals for a system supporting the forensic process [2]:

Interactivity. The potential damage that an attacker can wreak inside an organization grows quickly as a function of time, making fast detection and containment a vital concern. Further, a system's interactivity greatly affects the productivity of an analyst [16]. We thus desire replies to queries to begin arriving within a second or so.

Scalability. The volume of data to archive and process exceeds the capacity of single-machine deployments. A fundamental challenge lies in devising a distributed architecture that scales with the number of nodes in the system, as well as maximally utilizes the cores available in each node.

Expressiveness. Representing arbitrary activity requires a richly typed data model to avoid losing domain-specific semantics when importing data. Similarly, the system should expose a high-level query language to enable analysts to work within their domain, rather than spending time translating their workflows to lower-level system idiosyncrasies.

In this work, we develop a system for network forensics and incident response that aims to achieve these goals. We present the design and implementation of VAST (*Visibility Across Space and Time*), a unified storage platform that provides: (*i*) an expressive data model to capture descriptions of various forms of activity; (*ii*) the capability to use a single, declarative query language to drive both post-facto analyses and detection of future activity; and (*iii*) the scalability to support archiving and querying of not just log files, but a network's *entire activity*, from high-level IDS alerts to raw packets from the wire.

The key to VAST's power concerns providing the necessary performance to support both very high data volumes (100,000s of events/sec) and interactive queries against extensive historical data. VAST features an entirely asynchronous architecture designed in terms of the actor model [25], a message-passing abstraction for concurrent systems, to fully utilize all available CPU and storage resources, and to transparently scale from single-node to cluster deployments. To support interactive queries, VAST relies extensively on bitmap indexes that we adapt to support its expressive query language.

Our evaluations show that on a single machine VAST can ingest 100 K events/sec for events with 20 fields, reflecting an input rate of 2 M values/sec. Moreover, distributed ingestion allows for spreading the load over numerous system entry points. Users receive a "taste" of their results typically within 1 sec. This first subset helps users to quickly triage the relevance of the result and move on with the analysis by aborting or modifying the current query. We also show that VAST, with its unified approach, can effectively serve as a high-volume packet bulk recorder.

We structure the rest of the paper as follows. In §2 we summarize related work. We present the architecture of VAST in §3 and our implementation in §4. In §5 we evaluate VAST and assess its aptness for the domain. Finally, we conclude in §6.

2 Related Work

Data Warehouses. VAST receives read-only data for archiving, similar to a data warehouse. Dremel [40] stores semi-structured data *in situ* and offers an SQL interface for ad-hoc queries with interactive response. Dremel's query executor forms a tree structure where intermediate nodes aggregate results from their children. VAST generalizes this approach with its actor-based architecture to both data import and query execution.

Succinct [1] also stores data *in situ*, but in compressed flat files that do not require decompression when searched. Internally, Succinct operates on suffix trees and therefore supports point, wildcard, and lexicographical lookup on strings. Other data types (e.g., arithmetic, compound) require transformations into strings to maintain a lexicographical ordering. Succinct exhibits high preprocessing costs and modest sequential throughput, rendering it inapt for high-volume scenarios. When the working set fits in memory, Succinct offers competitive performance, but not when primarily executing off the filesystem.

ElasticSearch [17] is a distributed, document-oriented database built on top of Apache Lucene [36], which provides a full-text inverted index. ElasticSearch hides Lucene behind a RESTful API and a scheme to partition data over a cluster of machines. VAST uses a similar deployment model, but instead provides a semi-structured data model and internally relies on different indexing technology more amenable to hit composition and iterative refinements.

Network Forensics. NET-Fli [20] is a single-machine NetFlow indexer relying on secondary bitmap indexes. The authors also present a promising (though patented) bit vector encoding scheme, COMPAX. Instead of hand-optimizing a system for NetFlow records, VAST offers a generic data model. The separation between base data and indexes has also found application in similar systems [51], with the difference of relying on a column store for the base data instead of a key-value store. The existing systems show how one can design a single-machine architectures, whereas we present a design that transparently scales from single-machine to cluster deployments.

The Time Machine [38] records raw network traffic in PCAP format and builds indexes for a limited set of packet header fields. To cope with large traffic volumes, the Time Machine employs a *cutoff* to cease recording a connection's packets after they exceed a size threshold. The system hard-codes the use of four tree indexes over the connection tuple, and cannot reuse its indexes across restarts. Similarly, NetStore [22], pcapIndex [19], and FloSIS [33] offer custom architectures geared specifically towards flow archival. VAST represents a superset of bulk packet recorders: it supports the same cutoff functionality, and packets simply constitute a particular event type in VAST's data model.

The GRR Rapid Response framework [11] enables live forensic investigations where analysts push out queries to agents running on end-hosts. A NoSQL store accumulates the query results in a central location, but GRR does not feature a persistence component to comprehensively archive end-host activity over long time periods. VAST can serve as a long-term storage backend for host-level data, which allows analysts to query both host and network events in a unified fashion.

Finally, existing aggregators such as Splunk [49] operate on unstructured, high-level logs that lack the semantics to support typed queries, and are not designed for storing data at the massive volumes required by lower-level representations of activity. Splunk in particular cannot dynamically adapt its use of CPU resources to change in workload.

Distributed Computing. The MapReduce [14] execution model enables arbitrary computation, distributed over a cluster of machines. While generic, MapReduce cannot deliver interactive response times on large datasets due to the full data scan performed for each job. Spark [58] overcomes this limitation with a distributed in-memory cluster computing model where data is efficiently shared between stages of computation. However, for rapid response times, the entire dataset must reside preloaded in memory. But analysts can rarely define a working set *a priori*, especially for spatially distant data, which can result in thrashing due to frequent loading and evicting of working sets from memory. We envision VAST going hand-in-hand with Hadoop or Spark, where VAST quickly finds a tractable working set and then hands it off to a system well-suited for more complex analysis.

3 Architecture

To support flexible deployments on large-scale clusters, as well as single machines while retaining a high degree of concurrency, we designed VAST in terms of the actor model [25]. In this model, concurrent entities ("actors") execute independently and in parallel, while providing local fault isolation and recovery. Using unique, location-independent addresses, actors communicate asynchronously solely via message passing. They do not share state, which prevents data races by design.

A related model of computation is communicating sequential processes (CSP) [26] in which processes communicate via synchronous channels. As a result, the sender blocks until the receiver has processed the message. This creates a tighter coupling compared to the asynchronous fire-and-forget semantics of actor messaging. CSP emphasizes the channel while the actor model its endpoints: actors have a location-independent address whereas processes remain anonymous. In the context of distributed systems, the focus on endpoints provides a powerful advantage: the actor model contains a flexible failure propagation model based on monitors, links, and hierarchical supervision trees [4]. These primitives allow for isolating failure domains and implementing local recovery strategies, and thus constitute an essential capability at scale, where component failure is the norm rather than the exception. For these reasons, we deem the actor model a superior fit for our requirements.

We first present the underlying data model and the associated query language (§3.1), and then VAST's components and their structure (§3.2).

3.1 Data Model

VAST's data model consists of *types*, which define the physical interpretation of *data*. A type's *signature* includes a type name and type attributes. A *value* combines a type with a data instance. An *event* is a value with additional metadata, such as a timestamp, a unique identifier, and arbitrary key-value pairs. A *schema* describes the access structure of one or more types.

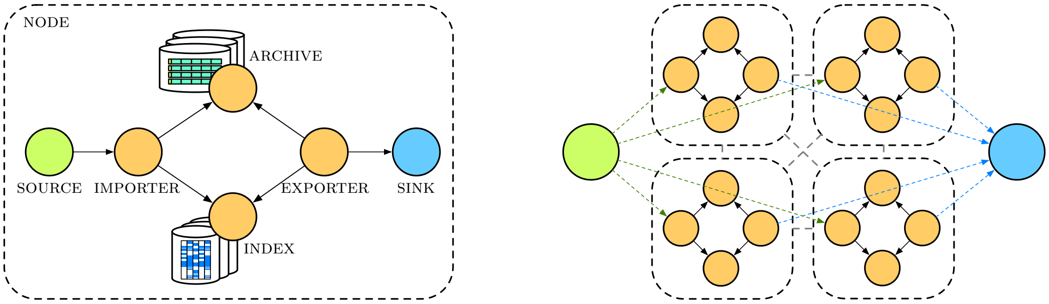
Boolean Expression	Symbol
Conjunction	E_1 && E_2
Disjunction	E_1 || E_2
Negation / Group	! E / (E)
Predicate	$LHS \circ RHS$
Relational Operator ○	**Symbol**
Arithmetic	<, <=, ==, !=, >=, >
Membership	in, !in
Extractor (LHS/RHS)	**Semantics**
:T	All values having type T
x.y.z	Value according to schema
&key	Event meta data
Types	**Examples**
bool	T, F
int / count / real	+42 / 42 / -4.2
duration / time	10ms / 2014-09-25
string	"foo", "b\x2Ar"
addr	10.0.0.1, ::1
subnet	192.168.0.0/24
port	80/tcp, 53/udp, 8/icmp
vector<T> / set<T>	[x, x, x] / {x, x, x}
table<T,U>	{(k,v), ..}

Table 1: VAST's query language.

VAST's type system includes *basic types* to represent a single value (booleans, signed/unsigned integers, floating-point, times and durations, strings, IPv4 and IPv6 addresses, subnets, ports), *container types* for bundled values (vectors, sets, tables), and *compound types* to create sequenced structures (records), where each named field holds a value (or nil if absent).

Query Language. VAST's query language supports filtering data according to boolean algebra. Table 1 lists the key syntactic elements. A query expression consists of one or more predicates connected with boolean AND/OR/NOT. A *predicate* has the form LHS ○ RHS, with ○ representing a binary relational operator. VAST supports arithmetic and membership operators. At least one side of the operator typically must be an *extractor*, which specifies the lookup aspect for the value, as follows.

Schema extractors refer to particular values in the schema. For example, in the predicate `http.method == "POST"`, `http.method` is a schema extractor, and `"POST"` is the value to match. *Meta extractors* refer to event metadata, such as `&name` to reference the event name and `&time` the event timestamp. For example, the predicate `&time > now - 1d` selects all events within



(a) Single-machine deployment. The NODE actor accommodates all key system actors.

(b) Cluster deployment. Multiple NODEs peer to form a cluster, which users access transparently.

Figure 1: VAST system architecture.

the last 24 hours. *Type extractors* leverage the strict typing in VAST to perform queries over all values having a given type. For example, the predicate `:addr in 10.0.0.0/8` applies to all IP addresses (any VAST value or record field with type `addr`).

To represent a log file having a fixed number of columns, VAST automatically transforms each line into a `record` whose fields correspond to the columns. VAST enforces type safety over queries and only forwards them to those index partitions with a compatible schema.

3.2 Components

From a high-level view, VAST consists of four key components: (*i*) *import* to parse data from a source into events and assign them a globally unique ID, (*ii*) *archive* to store compressed events and provide a key-value interface for retrieval, (*iii*) *index* to accelerate queries by keeping a partitioned secondary index referencing events in the archive, and (*iv*) *export* to spawn queries and relay them to sinks of various output formats. Each component consists of multiple actors, which can execute all inside the same process, across separate processes on the same host, or on different machines.

A NODE[1] represents a container for other actors. Typically, a NODE maps to a single OS-level process. Users can spawn components all within a NODE (Figure 1(a)) or spread them out over multiple NODEs (Figure 1(b)) to harness available hardware and network resources in the most effective way. NODEs can peer to form a cluster, using Raft [42] for achieving distributed consensus over global state accessible through a key-value store interface similar to etcd [18]. We refer to this globally replicated state as *meta store*. Each NODE has access to its own meta store instance which performs the fault-tolerant distribution of values. A typical cluster deployment exhibits a shared-nothing architecture, where each NODE constitutes a fully independent system by itself. In the following we discuss each of the four components in more detail.

Import. Data enters VAST via SOURCEs, each of which can parse various input formats. SOURCEs produce batches of events and relay them to IMPORTER, where they receive a unique monotone ID. Upon receiving the ID range and assigning them to the events, IMPORTER relays the batch to ARCHIVE and INDEX.

Each event represents a unique description of activity which analysts need to be able to unambiguously reference. This requires each event to have a unique identifier (ID) as meta data independent of its value. The ID also establishes a link between the archive and index component: an index lookup yields a set of IDs, each of which identify a single event in the archive. This yields the following requirements on ID generation: (*i*) *64 bits* to represent a sufficiently large number of events, but not larger since modern processors efficiently operate on 64-bit integers, (*ii*) *monotonicity* because the indexes we use are append-only data structures, and (*iii*) ID generation should also work in *distributed setups*.

The sequentiality requirement precludes approaches involving randomness, such as universally unique identifiers [32]. In fact, any random ID generation algorithm experiences collisions after $\approx \sqrt{N}$ IDs due to the birthday paradox. In combination with the 64-bit requirement, this would degenerate the effective space from 2^{64} to only $\sqrt{2^{64}} = 2^{32}$ IDs. Our approach uses a single distributed counter in the meta store. Requesting a range of N IDs translates to incrementing this counter by N, which returns a pair with the old and new counter value $[o,n)$ denoting the allocated half-open range with $n - o \leq N$ IDs. To avoid high latencies from frequent interaction with the meta store, IMPORTER keeps a local cache of IDs and only replenishes it when running low of IDs.

Archive. The ARCHIVE receives events from IMPORTER and stores them compressed on the filesystem.[2] To avoid frequent I/O operations for small amounts of

[1] We refer to particular actors in a SMALL CAPS font style.

[2] VAST supports LZ4 [37] and Snappy [48] for compression; both trade higher speeds for lower compression ratios.

data, ARCHIVE keeps event batches in a fixed-size memory buffer (by default 128 MB) before writing them to the filesystem. The buffer (which we term *segment*) keeps batches sorted by the ID of their first event. Because events have continuous IDs within a batch, this process ensures strictly monotonic IDs within a segment.

ARCHIVE exposes a key-value interface: queried with a specific ID, it returns a batch containing the ID. The alternative, returning the single matching event, only works for small requests, but would quickly bottleneck the messaging subsystem for moderate request volumes. Internally, ARCHIVE operates at the granularity of segments to further group event batches into larger blocks suitable for sequential filesystem I/O. ARCHIVE keeps an LRU cache of a configurable number of segments in memory. In the future, we plan to store data in a format also shareable with other applications, e.g., HDFS [46].

Index. By itself, ARCHIVE does not provide efficient access to data, since extracting events with specific properties would require a full scan of the archive. Therefore, VAST maintains a comprehensive set of secondary indexes, which we divide up in horizontal partitions as a unit of data scaling.

We chose bitmap indexes [41] because they provide an excellent fit for the domain. First, appending new values only requires time linear in the number of values in the new data, which is optimal and yields deterministic performance. Second, bitmap indexes have space-efficient representations that enable us to carry out bitwise operations without expanding them. Third, bitmap indexes *compose* efficiently: intermediate results have the form of bit vectors and combining them with logical operation translates into inexpensive bitwise operations. In §4.2 we describe bitmap indexes in more detail.

VAST's index consists of horizontal PARTITIONs, each of which manages an independent set of bitmap indexes. An *active* partition is mutable and can incorporate events, whereas a *passive* partition is an immutable unit which the scheduler manages during query processing. INDEX relays each arriving batch of events to the currently active PARTITION, which spawns a set of INDEXERs, one per event type. An INDEXER may internally further helper actors, e.g., record INDEXERs use one helper per field. In comparison to a relational database, this architecture provides a fine-grained concurrent equivalent to tables (INDEXERS) and their attributes. PARTITION relays each batch concurrently to *all* INDEXERs, each of which processes only the subset matching its type. The copy-on-write message passing semantics of the actor model implementation makes this an efficient operation (INDEXERs only need read-access to the events) while providing a high degree of parallelism.

Export. While the import component handles event ingestion, the export component deals with retrieving events through queries. There exists one EXPORTER per query, who sends its result to SINKs for rendering. VAST currently includes ASCII, JSON, PCAP [35], Bro [43], and Kafka [29] SINKs.

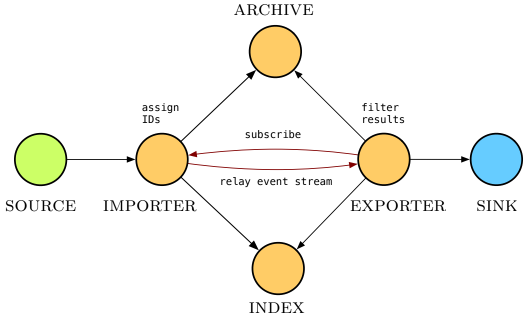


Figure 2: Continuous query architecture.

For *historical queries* over existing data, INDEX analyzes the abstract syntax tree (AST) of the query expression to determine the relevant PARTITIONS and constructs a schedule to process them sequentially. This process begins with *partition pruning*: the selection of relevant partitions for a query. To this end, VAST uses a "meta index," consisting of event meta data and event type information. The meta index has different requirements than the event indexes within a partition: it must tolerate immense throughput and update rates. We currently associate with each partition the time period its events spans and record the entire partition schema.

After pruning, INDEX relays the query to the remaining PARTITIONs, which then deconstructs the AST into its predicates to match them with INDEXERs. If necessary, PARTITION loads the INDEXER from the filesystem into memory. Upon performing the predicate lookup, INDEXERS send their hits back to their PARTITION, where they trigger a re-evaluation of the query expression. If the evaluation yields a change, PARTITION relays the *delta hits* from the last evaluation up to INDEX, which in turn forwards them to the subscribed EXPORTERs. As soon as the first hits arrive at an EXPORTER, extracting events can begin by asking ARCHIVE. When EXPORTER receives an answer in the form of a batch of events, it concurrently prefetches another batch proceeding with the "candidate check" to filter out false positives, which may be necessary for some indexes types (e.g., when using binning for floating point values, see §4.2). Finally, EXPORTER sends the matching results back to SINK. The process terminates after EXPORTER has no more unprocessed hits to extract.

VAST also supports *continuous queries* to subscribe to new results as they arrive. As we illustrate in Figure 2, EXPORTER can subscribe to a copy of the full incoming event feed at IMPORTER to filter out those events matching the query expression. Since IMPORTER and EXPORTER live in the same process this operation does not copy any data. In fact, a continuous query is effectively a candidate check, with the only difference that events now come from IMPORTER instead of ARCHIVE. The main challenge lies in efficiently performing this check. To this end, EXPORTER derives from the AST of the query expression a visitor performing the candidate check. EXPORTER constructs one visitor per unique event type on the fly, and dispatches to the relevant visitor efficiently through a hash table. For example, an expression may include the predicate `:addr in 10.1.1.0/24`. If the current event has no data of type `addr`, the visitor discards the predicate.

3.3 Distribution

When the amount of data exceeds the capacity of a single machine, VAST can distribute the load over multiple machines, as we show in Figure 1(b) for 4 peering NODEs.

Cluster Deployment. In case a SOURCE produces events at a rate that overloads a single system, it can load-balance its events over all NODEs. Alternatively, users can pin SOURCEs to a specific set of NODEs (e.g., if certain data should land on a more powerful system). The dual occurs during querying: the user decides on which NODEs to spawn an EXPORTER, each of which will relay its events to the same SINK. In the common case of round-robin load-balancing of input data, a user query results in spawning one EXPORTER on all NODEs.

Fault Tolerance. Coping with NODE failure concerns two aspects: data loss and query execution. To avoid permanent data loss, we assume a reliable, distributed filesystem, e.g., HDFS. Data loss can still occur during ingestion, when ARCHIVE and INDEX have not yet written their data to the filesystem. VAST minimizes this risk by writing data out as quickly as possible. When a NODE fails during query execution, another takes over responsibility of its persistent ARCHIVE and INDEX data, and re-executes the query on its behalf. EXPORTER can periodically checkpoint its state (consisting of index hits and event identifiers of results that have passed the candidate check) to reduce the amount of duplicate results.

4 Implementation

We now present some implementation highlights of the previously discussed architecture with a focus on actors (§4.1), bitmap indexes (§4.2), and queries (§4.3).

4.1 Actors

We implemented the components discussed in §3.2 with the C++ Actor Framework (CAF), a native implementation of the actor model [10].

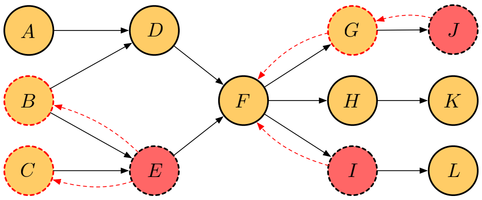


Figure 3: Flow-control implemented as back-pressure: overloaded nodes J, I, and E propagate their load status upstream such that data sources can throttle their sending rate.

Performance. CAF offers a high-performance runtime with a type-safe messaging layer that exhibits minimal memory overhead. The runtime can distribute actors dynamically, within the same process, across the network, or to GPUs. Within the same process, CAF uses copy-on-write semantics to enable efficient messaging. CAF's networking layer handles actor communication and routing *transparent* to the user: the runtime decides whether sending a message translates into a local enqueue operation into the recipient's mailbox, or whether a middleman serializes the message and ships it across the network.

CAF's copy-on-write messaging proves particularly valuable during the indexing process, where PARTITION sends the *same set of events* to each INDEXER without incurring a copy of the data. Although each INDEXER sees the full data feed, this method still runs faster than chopping up the input sequence and incurring extra memory allocations. Such a computation style resembles GPU programming where we make available data "globally" and each execution context picks its relevant subset.

Flow Control. CAF operates entirely asynchronously and does not block: immediately after sending a message an actor can dequeue a message from its mailbox. This makes it easy for data producers to overwhelm downstream nodes if not equipped with enough processing capacity. A naive reaction entails provisioning more buffer capacity at the edge so that the system can receive more messages. But without including the true bottlenecks in the decision, buffer bloat [31] only worsens the situation by introducing higher latency and jitter. Flow control attempts to prevent this scenario from happening: *back-pressure* signals overload back to the sender, *load shedding* reduces the accumulating tasks at the bottleneck, and *timeouts* at various stages in the data flow graph bound the maximum response time.

CAF currently does not support flow control. During data import, data producers (SOURCEs) can easily overload downstream components (ARCHIVE and INDEX). To throttle the sending rate of SOURCEs, we implemented a simple back-pressure mechanism: when an actor becomes overloaded, it sends an overload message to all its registered upstream components, which either propagate it further, or if being the data producer themselves, throttle their sending rate (see Figure 3). When an overloaded actor becomes underloaded again, it sends an underload message to signal upstream senders that it can process data again. This basic mechanism works well to prevent system crashes due to overloads, but does not help at absorb peak input rates. To prevent data loss of non-critical, latency-insensitive events, queueing brokers that spill to the filesystem (e.g., Kafka [29]) at the edges help smoothing the data arrival rate to facilitate resource provisioning for the average case. We are currently working with the CAF developers to integrate various forms of flow control deeper into the runtime.

4.2 Composable Indexing

VAST exclusively relies on bitmap indexes to accelerate queries. We begin in §4.2.1 with briefly summarizing existing work, which we then rely on in §4.2.2 to define composable higher-level index types.

4.2.1 A Unified Indexing Framework

Traditional tree and hash indexes [6, 30] provide a quick entry point into base data, but they do not compose efficiently for higher-dimensional search queries. Inverted and bitmap indexes avoid this problem by adding an extra level of indirection: instead of looking up base data directly, they operate on the IDs of the base data, allowing for combining results from multiple index lookups via set operations. Consider the event values $x^{(\alpha)}$ where x represents a numeric value and α its ID, e.g., $1^{(0)}$, $3^{(1)}$, $1^{(2)}$, $2^{(3)}$, and $1^{(4)}$. An inverted index represents the events as a mapping from values to *position lists*: $1 \rightarrow \{0,2,4\}$, $2 \rightarrow \{3\}$, and $3 \rightarrow \{1\}$. A bitmap index is an isomorphic data structure where the position lists have the form of *bit vectors*:[3] $1 \rightarrow \langle 10101 \rangle$, $2 \rightarrow \langle 00010 \rangle$, and $3 \rightarrow \langle 01000 \rangle$. When the distinction does not matter, we use the term *identifier set* to mean either position list or bit vector.

There exist hybrid schemes which combine both index types in a single data structure [7], but VAST currently implements its algorithms only in terms of bitmap indexes, where operations from boolean algebra (set, intersection, complement) naturally map to native CPU instructions. In general, an index I provides two basic primitives: adding new values and looking up existing ones under a given predicate. To add a new value $x^{(\alpha)}$, the index adds α to the identifier set for x. A lookup under a predicate $I \circ x$ retrieves the identifier set $S = \{\alpha \mid x^{(\alpha)} \in I\}$. The *size* of and index $|I| = N$ represents the number of values entered and the *cardinality* $\#I = C$ the number of distinct values.

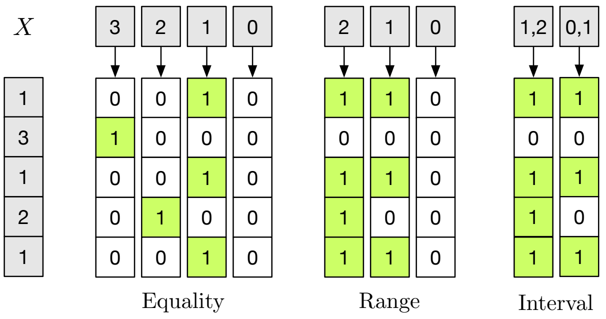


Figure 4: Equality, range, and interval encoding exemplified using bitmap indexes.

Our example has $N = 5$ and $C = 3$. In the following, we sketch key concepts that affect the inherent space-time trade-off during the implementation of indexes, which include binning, coding, compression, and composition.

Binning reduces the cardinality of an index by grouping values into bins or rounding floating-point values to a certain decimal precision. For example, we could create bins $[1,2] \rightarrow \langle 10111 \rangle$ and $[3,4] \rightarrow \langle 01000 \rangle$, which reduces the cardinality of the index to $C = 2$. The surjective nature of binning introduces false positives and therefore requires a *candidate check* with the base data to verify whether a certain hit qualifies as an actual result. A candidate check can easily dominate the entire query execution time due to materializing additional base data (high I/O costs) and extra post-processing. Therefore, choosing an efficient binning strategy requires careful tuning and domain knowledge, or advanced adaptive algorithms [44, 47].

Encoding determines how an index maps values to identifier sets. We show in Figure 4 the three major existing encoding schemes for a fixed cardinality $C = 4$, which can represent values 0–3. *Equality encoding* associates each value with exactly one identifier set. This scheme reflects our running example and consists of exactly C identifier sets. *Range encoding* associates each value $x^{(\alpha)}$ with a range of $C - 1$ identifier sets such that an ID α lands in i sets where $x \leq i$. We can omit the last identifier set because $x \leq C - 1$ holds true for all possible values. *Interval encoding* splits the index into $\lceil \frac{C}{2} \rceil$ overlapping slices, each of which covers half of the values. In our example, we have two the intervals $[0,1]$ and $[1,2]$.

Compression algorithms for bit vectors typically use variations of run-length encoding, which support bitwise operations without prior decompression. There exist numerous algorithms: BBC [3], WAH [56], COMPAX [20], CONCISE [12], WBC/EWAH [57, 34], PLWAH [15], DFWAH [45], PWAH [53], VLC [13], and VAL [24]. We chose EWAH for VAST because when we began our project software patents covered (and still do) the other attractive candidates (WAH, PLWAH, COMPAX), which would have prevented us from releasing our project as

[3] The literature often uses the term "bitmap" to refer to a bit vector, i.e., a sequence of bits. We use "bit vector" instead to avoid confusion between "bitmap" and "bitmap index."

open-source software. EWAH also trades space for time: while exhibiting a slightly larger footprint, it executes faster in certain conditions [23] because it can skip entire blocks of words.

Multi-component indexes combine several individual index instances (which might use different approaches) such that each covers a disjoint partition of the value domain. Doing so provides an exponential reduction in space by decreasing the size of the value domain by a multiplicative factor for each component. We can decompose a value x into k components $\langle x_k, \ldots, x_1 \rangle$ by representing it with respect to a fixed *base* (or radix) $\beta = \langle \beta_k, \ldots, \beta_1 \rangle$: $x = \sum_{i=1}^{k} x_i \beta_i$, where $x_i = \lfloor x / \prod_{j=1}^{i-1} \beta_j \rfloor \bmod \beta_i$, for all $i \in \{1, \ldots, k\}$. This decomposition scheme directly applies to the index structure as well: a multi-component index $K^\beta = \langle I_k, \ldots, I_1 \rangle$ consists of k indexes, where each I_i covers a total of β_i values. A base is *uniform* if $\beta_i = \beta_j$ for all $i \neq j$. A uniform base with $\beta_i = 2$ for all $1 \leq i \leq k$ yields the *bit-sliced index* [55], because each x_i can only take on values 0 and 1. We denote this special case by $\Theta^k = K^\beta$ where $|\beta| = k$ and $\beta_i = \beta_j = 2$ for all $i \neq j$. Further, we define $\Phi^w = K^\beta$ where $\prod_{i=1}^{k} \beta_i \leq 2^w$ as an index which supports up to 2^w values.

For example, consider a two-component index $K^\beta = \langle I_2, I_1 \rangle$ with $\beta = \langle 10, 10 \rangle$, which supports 100 distinct values. Appending a value $x^{(\alpha)} = 42$ involves first decomposing it into $\langle 4, 2 \rangle$, and then appending $4^{(\alpha)}$ to I_2 and $2^{(\alpha)}$ to I_1. Looking up the value $x = 23$ begins with decomposing x into $\langle 2, 3 \rangle$, and then proceeds with computing $I_2 = 2 \wedge I_1 = 3$. The final step resolves each component lookup according to its encoding scheme. Operators other than $\{=, \neq\}$ require more elaborate lookup algorithms [8, 9], which we lay out in greater detail separately [52].

4.2.2 Higher-Level Indexing

For each type in VAST's data model, there exist different requirements derived from the desired query operations. For example, numeric values commonly involve inequality comparisons and IP address lookups top-k prefix search. Different lookup operations require different index layouts.

Per §3.1, a *value* consists of a *type* and corresponding *data*. A value can exhibit no data, in which case it only carries type information. We define the *value index* $\mathbb{V} = \langle N, D \rangle$ as a composite structure with a *null index* N to represent whether data is null (implemented as single identifier set), and a *data index* D to represent a type-specific index, whose instantiations we describe next.

Integral Indexes. The *boolean index* $\mathbb{B} = \langle S \rangle$ for type `bool` consists of a single identifier set, where $\alpha \in S$ implies $x^{(\alpha)} = \texttt{true}$.

For types `count` and `int`, the challenge lies in both supporting lookups under $\{<, \leq, =, \neq, \geq, >\}$, as well as representing 2^{64} distinct values. To address the high cardinality challenge, we use a multi-component index Φ^{64}. We found that a uniform base 10 works well in practice. To support the desired arithmetic operations, we use range coding, which supports both equality and inequality lookups efficiently.

Signed integers introduce a complication: we cannot map negative values to array indices during encoding, and only positive numbers work with value decomposition. For a w-bit signed integer, we therefore introduce a "bias" of 2^{w-1}, which shifts the smallest value of -2^{w-1} to 0 in the unsigned representation. This allows us to use the same index type for signed an unsigned integers internally. Thus, we define the *count index* as $\mathbb{C} = \Phi^{64}$ and the *integer index* as $\mathbb{I} = \Phi^{64}$ with the aforementioned bias.

Floating-Point Index. Type `real` corresponds to a IEEE 754 double precision floating point value [28], which consists of one sign bit, 11 bits for the exponent, and 52 bits for the significand. Consequently, we construct the *real index* $\mathbb{F} = \langle S, E, M \rangle$ as a boolean index $S = \mathbb{B}$ for the sign, a bit-sliced index $E = \Theta^{11}$ for the exponent, and a bit-sliced index $M = \Theta^{52}$ for the significand. Varying the number of bits in E and M allows for trading space for precision without the need to round to a specific decimal point.

Temporal Indexes. VAST represents `duration` data as a 64-bit signed integers in nanosecond resolution, which can represent ±292.3 years. Since `duration` and `int` are representationally equal, the *duration index* $\mathbb{D} = \mathbb{I}$ directly maps to an integer index.

The type `time` describes a fixed point in time, which VAST internally expresses as `duration` relative to the UNIX epoch. Thus, the *time index* $\mathbb{T} = \mathbb{D}$ is representationally equal to the duration index.

String Index. Existing string indexes rely on a dictionary to map each unique string value to a unique numeric value [50]. However, constructing a space-efficient dictionary poses a challenge in itself [39, 27]. Moreover, this design only supports equality lookups naturally: for substring search, one must search the dictionary key space first to get the identifier sets, and then perform the lookup for each identifier set, and finally combine the result. Instead of using a stateful dictionary, one can rely on hashing to compute a unique string identifier [51]. The possibility of collisions now requires a candidate check. While space-optimal due to the absence of a dictionary, and time-efficient due to fast computation, this approach does not support substring search.

We propose a new approach for string indexing that supports both equality and substring search, yet operates in a stateless fashion without dictionary. Our *string index* $\mathbb{S} = \langle \phi, \kappa_1, \ldots, \kappa_M \rangle$ consists of an index $\phi = K^\beta$ for the string length, plus M indexes $\kappa_i = \Phi^8$ per character where

M is the largest string added to the index. When representing the character-level indexes κ_i as a bit-sliced index Θ^8, we obtain efficient case-insensitive (substring) search for ASCII-encoded strings. This works because only the 6th bit determines casing in ASCII, and by simply omitting the corresponding identifier set Θ^8_6 during lookup, case-insensitive search executes *faster* than case-sensitive search. We plan on supporting search via a subset of regular expressions by compiling a pattern into an automaton performing a sequence of per-character lookups. Likewise this structure lends itself to similarity search, e.g., via edit-distance.

This design works well for bounded, non-uniform string data, such as URLs or HTTP user agents. For other workloads we fall back to hashing in combination with *tokenization*. This preprocessing step splits a string according to a pattern (e.g., whitespace for text, '/' for URIs, etc.), which then creates a set of multiple smaller strings. Adaptively switching between the index types presents an interesting opportunity for future work, e.g., by inspecting both the nature of user queries as well as inferring the data distribution.

Network Indexes. IP addresses constitute a central data type to describe endpoints of communicating entities. The most common operation on IP addresses consists of top-k prefix search, e.g., $I \in$ `192.168.0.0/24` or $I \notin$ `fd00::/8`. We can consider equality lookup as a special case when $k = 32$ and $k = 128$ for IPv4 and IPv6, respectively. There exists a standardized scheme to embed a 32-bit IPv4 address inside a 128-bit IPv6 address [5]: set the first 96 bits to 0 and copy the IPv4 address in the last 32 bits. This yields the *address index* $\mathbb{A} = \Theta^{128}$ where each bit in the address corresponds to one identifier set in the index.

The `subnet` type consists of a network address and a prefix. Typical queries involve point lookups of IP addresses (e.g., `192.168.0.42` $\in I$), and subset relationships to test whether one subnet contains another (e.g., `192.168.0.0/28` $\subseteq I$). The *subnet index* $\mathbb{U} = \langle \mathbb{A}, P \rangle$ consists of an address index $\mathbb{A}$ and a single equality-encoded index P for the prefix.

The `port` type consists of a 16-bit number and a transport protocol type. The *port index* $\mathbb{P} = \langle \Phi^{16}, T \rangle$ consists of an index for the 16-bit bit port number and a single equality-encoded index T for the different port types.

Container Indexes. Container types include `vector`, `set`, and `table`. A container contains a variable number of elements of a homogeneous type, unlike `records`, which allows for fixed-length heterogeneous data with named fields. For example, a `set` describes DNS lookup results, where a single host name has associated multiple A records. We support cardinality and subset queries on containers.

The design of string indexes generalizes to containers. Let M denote the maximum number of elements in a container. We define the *vector index* $\mathbb{X}^V$ and *set index* $\mathbb{X}^S$ both as $\langle \phi, \mathbb{V}_1, \ldots, \mathbb{V}_M \rangle$: an index $\phi = K^\beta$ for the container size and M value indexes $\mathbb{V}_i$. The *table index* $\mathbb{X}^T = \langle \phi, X_1, \ldots, X_M, Y_1, \ldots, Y_M \rangle$ consists of an index for the size ϕ, a sequence of value indexes $X_i = \mathbb{V}$ for the table keys, and a sequence of value indexes $Y_i = \mathbb{V}$ for the table values. Tables support key lookup, value lookup, and checking for specific mappings.

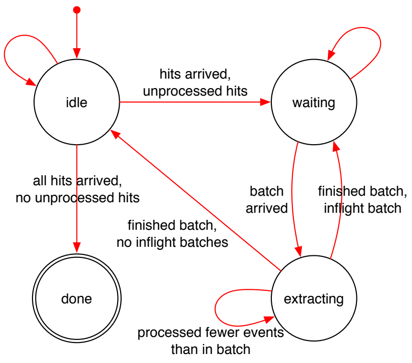


Figure 5: The QUERY state machine.

4.3 Queries

The actor model provides an apt vehicle to implement a fully asynchronous architecture, which enables VAST to deliver interactive response times. We illustrate how this applies to query processing in the following.

Finite state machines. We found that finite state machines (FSMs) prove an indispensable mechanism to ensure correct message handling during query execution. Recall that NODE spawns an EXPORTER for each query to bridge ARCHIVE and INDEX. We implemented EXPORTER as a finite state machine (see Figure 5), which begins in `idle` state. Upon receiving new hits EXPORTER asks ARCHIVE for the corresponding batches and transitions into `waiting`. As soon as the first batch arrives, it transitions into `extracting`, from where a user can selectively control it to fetch specific results. By letting the user drive the extraction, VAST does not consume resources unless needed.

Predicate-level caching. To speed-up related queries, INDEX maintains a predicate cache. If hits for the expression `A || B` exists already, then a new expression `A && D` only requires looking up `D`. This makes iterative query styles viable, where the analyst keeps on refining a filter until having pin-pointed the desired information.

Evaluating expressions. When INDEX receives a query consisting of multiple predicates, VAST evaluates them concurrently. Recall from §3.2 that during a historical query INDEXERs send their hits back to PARTITION, where they trigger an evaluation of the expression. If the evaluation yields new hits (i.e., a bit vector with new

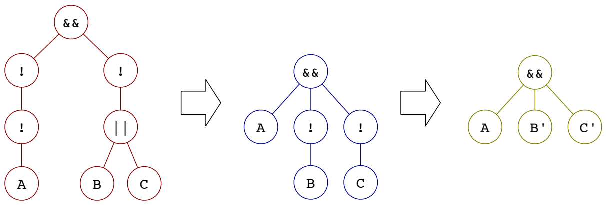


Figure 6: Expression normalization: negation normal form (NNF) and negation absorbing. The expression $\overline{\overline{A}} \wedge \overline{B \vee C}$ first becomes $A \wedge \overline{B} \wedge \overline{C}$. Thereafter, we can absorb negations further and reduce the intermediate expression to $A \wedge B' \wedge C'$, e.g., if $B = \overline{I < x}$, then $B' = I \geq x$.)

1-bits), PARTITION forwards them to INDEX, which in turn relays them to EXPORTER.

To minimize latency and relay hits as soon as possible, we normalize queries to negation normal form (NNF), which eliminates double-negations and pushes negations inwards to the predicate level. We also absorb remaining negations into the predicates, which is possible because each operator has a complement (e.g., $<$ and $\geq$). Figure 6 illustrates the normalization procedure. The absence of negations, aside from saving an extra complement operation, has a useful property: a 1-bit will never turn to 0 during evaluation.

To understand this benefit, consider a predicate A which decomposes into n sub-predicates. This may occur for predicates of the form `:addr in 172.16.0.0/16`, where the type extractor acts as a placeholder resolving to n concrete schema extractors. When PARTITION sends A to the n INDEXERS, they report their hits asynchronously as soon as they become available. PARTITION continuously re-evaluates the AST for new arriving H_i, until having computed $A = H_1 \vee \cdots \vee H_n$. As soon as a re-evaluation yields one or more new 1-bits, PARTITION relays this delta upstream to INDEX. If we kept the negation $\overline{A}$, we would to wait for *all n* hits to arrive in order to ensure we are not producing false positives. Without negations, we can relay this change immediately since a 1-bit cannot turn 0 again in a disjunction.

5 Evaluation

We evaluate our implementation in terms of throughput (§5.1), latency (§5.2), and storage requirements (§5.4). We performed measurements with two types of inputs: synthetic workloads that we can precisely control, and real-world network traffic. For the former, we implemented a benchmark SOURCE that generates input for VAST according to a configuration file. The SOURCE generates all synthetic data in memory to avoid adding I/O load. For real-world input, we use logs from Bro and raw PCAP traces. For the latter, VAST functions as a flow-oriented bulk packet recorder.

VAST comprises 36,800 lines of C++14 code (excluding whitespace and comments), plus 6,700 line of C++ unit tests verifying the system's building blocks and basic interactions. We expand on these checks with an end-to-end test of whether the entire pipeline—from import, over querying, to export—yields correct results. For validation, we processed our ground truth (Bro logs and PCAP traces) separately and cross-checked against the query results VAST delivers. We found full agreement.

We conducted our single-machine evaluation experiments on a 64-bit FreeBSD system with two 8-core Intel Xeon E5-2650 CPUs with 128 GB of RAM and four 3 TB SAS 7.2 K disks (RAID 10 with 2 GB of cache). Our dataset encompasses a full-packet trace from the upstream link at the International Computer Science Institute (ICSI), containing 10 M packets over a 24-hour window on Feb. 24, 2015. We further use 3.4 M Bro connection logs derived from this trace. For our cluster experiments, we use 1.24 B Bro connection logs (152 GB), split into N slices for N worker nodes, with N ranging from 1 to 24. Each worker node runs FreeBSD 10 on a system with two 8-core Intel Xeon E5430 CPUs with 12 GB of RAM and 2 x 500 MB SATA disks. An additional machine with two Xeon X5570 CPUs and 24 GB performs the slicing. The machines share a 1 GE network link.

5.1 Throughput

One key performance metric represents the rate of events that VAST can ingest. Recall the data flow: SOURCEs parse and send input to a system entry point, an IMPORTER, which dispatches the events to ARCHIVE and INDEX. Because we can spawn multiple SOURCEs for arbitrary subsets of the data, we did not optimize SOURCEs at this stage in the development, nor ARCHIVE since it merely sequentially compresses events into fixed-size chunks and writes them out to the filesystem. Instead, we concerned ourselves with achieving high performance at the bottleneck: INDEX, which performs the CPU-intensive task of building bitmap indexes.

Macro Benchmark. For the ingestion benchmark, we configured a batch size of 65,536 events at SOURCE, after observing that greater values entail slightly poorer performance and higher variance (we tested up to 524,288 events per batch).

Figure 7 shows the event rates for three data formats (Bro, PCAP, and a benchmark test) at SOURCE, ARCHIVE, and INDEX as a function of number of cores provided to CAF's scheduler in the single system setup. The y-axis shows the throughput in events per second; note the log scale. As mentioned above, by design SOURCE and ARCHIVE exhibit a fairly constant throughput rate. The highly concurrent architecture complicates measurements of aggregate throughput at INDEX, because there

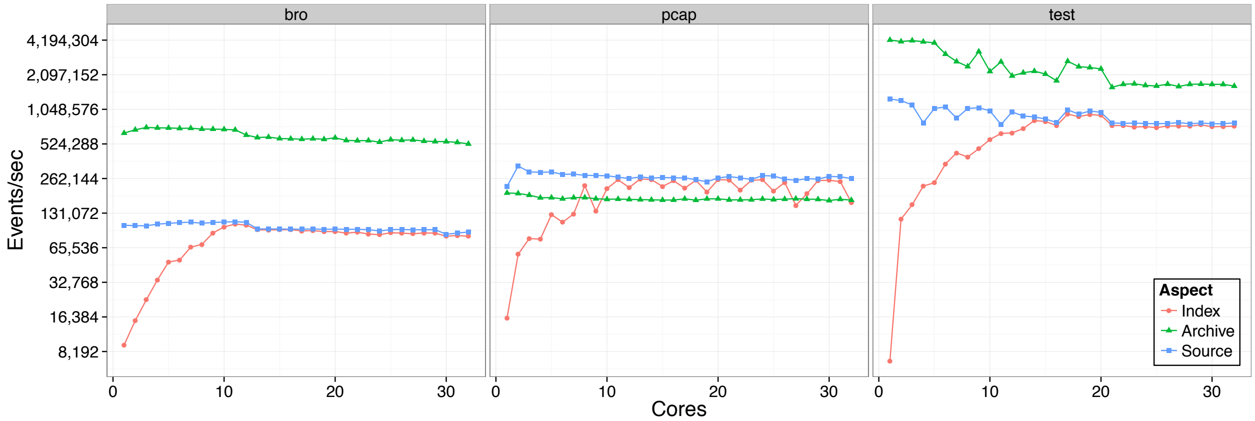


Figure 7: Throughput in events per second as a function of the number of cores for three data types: Bro connection logs, PCAP traces, and a test source generating synthetic events.

Label	Results	Query	Description
A	374	resp_h == 2001:7fe::53	Connections to a specific IPv6 address
B	942	(duration > 1000s || resp_bytes > 40000) && service == "dns"	Anomalous DNS / zone transfers
C	13	orig_h in 192.150.186.0/23 && orig_bytes > 10000 && service == "http"	Outgoing HTTP requests > 10 KB (exfiltration)
D	3	duration > 1h && service == "ssh"	Long-lived SSH sessions
E	969,092	conn_state != "SF"	TCP sessions lacking normal termination
F	4812	:addr in 192.150.186.0/23 && :port == 3389/?	All RDP involving ICSI connections
G	1,077	:addr in 192.150.186.0/23 && :port == 3389/?	Same as above, but applied to PCAP trace
H	34	&time > 2015-02-04+10:00:00 && &time < 2015-02-04+11:00:00 && ((src == 77.255.19.163 && dst == 192.150.187.43 && sport == 49613/? && dport == 443/?) || (src == 192.150.187.43 && dst == 77.255.19.163 && sport == 443/? && dport == 49613/?))	Extract all packets from a single connection specified by its 4-tuple and restricted to a one-hour time window
I	187,015	&time > 2015-02-04+10:00:00 && &time < 2015-02-04+11:00:00 && :addr == 192.150.187.43	All traffic from a single machine within a one-hour window

Table 2: Test queries for single-machine throughput and latency evaluation. The top 6 queries run over Bro connection logs and the bottom 3 over a PCAP trace.

exist multiple running INDEXERs, but not all start and finish at the same time. Therefore we compute throughput at INDEX as the number of events processed between start and end of the measurement. Consequently, the throughput can never exceed the input rate of SOURCE. We observe that the indexing rate approaches the input rate for all sources at around 10 cores. Giving CAF's work-stealing scheduler, more cores yield no further improvement; in fact, performance decreases slightly. We presume this occurs to due thrashing since CAF does not pin the worker threads to a specific core, which increases context switches and cache evictions.

VAST parses Bro events at a rate of roughly 100 K events per second, with each event consisting of 20 different values, yielding an aggregate throughput of 2 M values per second. For PCAP, events consist of only the 4-tuple plus the full packet payload; the latter do not need indexing. VAST can read at around 260 K packets per second with `libpcap`. Since ARCHIVE does not skip the payload, it cannot keep up with the input rate. This suggests that we need to parallelize this component in the future, which can involve spawning one COMPRESSOR per event batch to parallelize the process. With our test SOURCE, INDEX converges to the input rate at around 14 cores, and we observe input rates close to 1 M events per second. We conclude that VAST meets the performance and scalability goals for data import on a single machine: the system scales up to the point of the input rate after 10-14 cores.

Micro Benchmark. To better understand where VAST spends its time, we instrumented CAF's scheduler to get fine-grained, per-actor resource statistics. This involved bracketing the job execution with resource tracking calls (`getrusage`), i.e., we only measure actor execution and leave CAF runtime overhead, mostly out of the picture.

In Figure 8, we plot user versus system CPU time for all key actors. Each point represents a single actor instance, with its size scaled to the utilization: user plus system CPU time divided by wallclock time. Note the log scale on both axes. In the top-right corner, we see ARCHIVE, which spends its time compressing events (user) and writing chunks to disk (system). Likewise, INDEX appears nearby, which manages primarily PARTITIONs and builds small "meta indexes" based on time to quickly identify which PARTITION to consider during a query. The bulk of the processing time spreads over numerous INDEXERs,

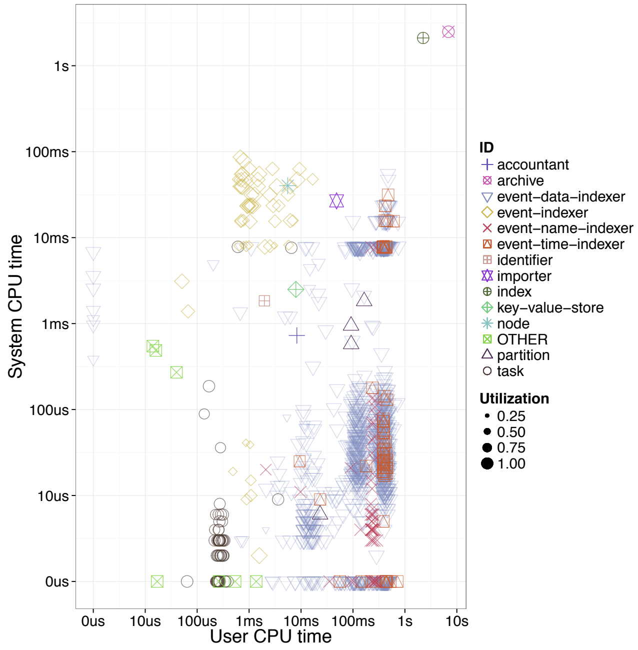


Figure 8: User versus system CPU time for key actors. Each point represents a single actor instance, with point size scaled to utilization: user plus system CPU time divided by wallclock time.

which we can see accumulating on the right-hand side, because building bitmap indexes is a CPU-bound task.

5.2 Latency

Query response time plays a crucial role in assessing the system's viability. VAST spawns one EXPORTER per query, which acts as a middleman receiving hits from INDEX and retrieving the corresponding compressed chunks of events from ARCHIVE. This architecture exhibits two interleaving latency elements: the time (*i*) from the first to the last set of hits received from INDEX, and (*ii*) from the first to the last result sent to a SINK after a successful candidate check.

To evaluate these latency components, we use the set of test queries given in Table 2, which a security operator for a large enterprise confirmed indeed reflect common searches during an investigation.

Query Pipeline. Figure 9 illustrates the latency elements seen over the test queries. For all queries, we ran VAST with 12 cores and a batch size of 65,536. The first red bar corresponds to the time it took until EXPORTER received the first set of hits from INDEX. The green bar shows the time until EXPORTER has sent the first result to its SINKs. We refer to this as "taste" time, since from the user perspective it represents the first system response. The blue bar shows the time until EXPORTER has sent the full set of results to its SINK. The black transparent box corresponds to the time when INDEX finished the computation of hits. Finally, the crosses inside the bar correspond to points in time when hits arrive, and the circles to the times when EXPORTER finishes extracting results from a batch of events.

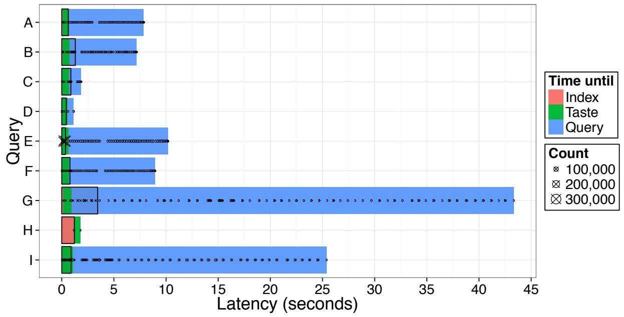


Figure 9: Query pipeline reflecting various stages of single-node execution. The first stage (Index) may appear absent because it can take too little time to manifest in the plot.

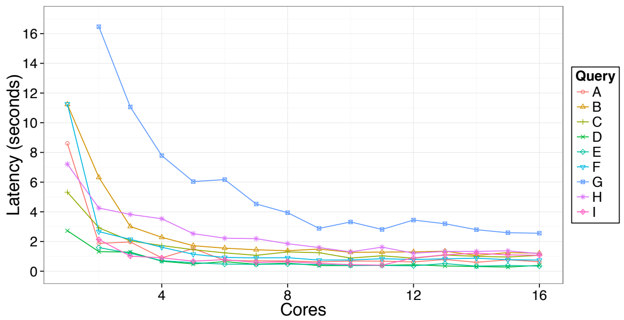


Figure 10: Index latency (full computation of hits) as a function of cores.

We see that extracting results from ARCHIVE (blue bar) accounts for the largest share of execution time. Currently, this time is a linear function of the query selectivity, because EXPORTER does not perform extraction in parallel. We plan to improve this in the future by letting EXPORTER spawn a dedicated helper actor per arriving batch from ARCHIVE, allowing for concurrent sweeps over the candidates. Alternatively, we could offload more computation into ARCHIVE. Selective decompression algorithms [21] present an orthogonal avenue for further improvement.

Index. VAST processes index lookups in a continuous fashion, with first hits trickling in after a few 100 msecs. Figure 10 shows that nearly all index lookups fully complete within 3 seconds once we use more than 4 cores. For query G, we observe scaling gains up to 10 cores. This particular query processes large intermediate bit vectors during the evaluation, which require more time to combine.

Overall, we find that VAST meets our single-machine performance expections. In particular, we prioritized ab-

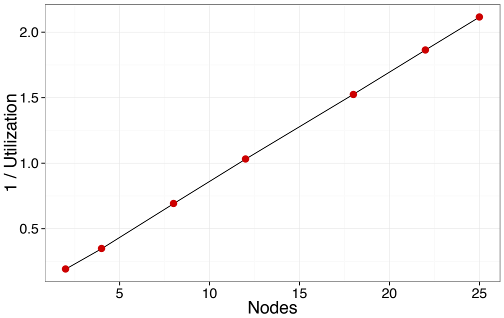


Figure 11: Per-node CPU utilization during ingestion.

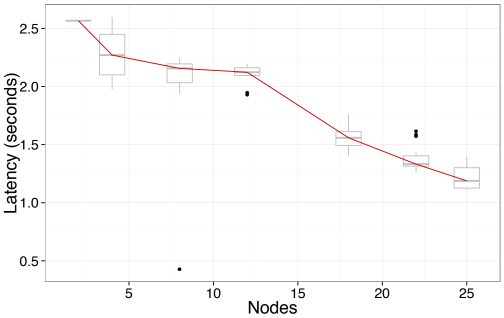


Figure 12: Index completion latency as function of nodes.

straction to performance in our implementation, and have not micro-optimized code bottlenecks (such as via inspecting profiler call graphs). Given that each layer of abstraction—from low-level bit-wise operations to high-level concurrency constructs—comes at the cost of performance, we believe that future tuning efforts hold promise for even further gains.

5.3 Scaling

In addition to single-machine benchmarks, we analyze how VAST scales over multiple machines in a cluster setting, as this will constitute the only viable deployment model for large sites exhibiting copious amounts of data.

Ingestion. Our first measurement concerns quantifying how CPU load during event import varies as a function of cluster nodes. To this end, we ingest 1.24 B Bro connection logs by load-balance them over the cluster NODEs in batches of 65 K. That is, as in Figure 1(b), a SOURCE on a separate machine parses the logs and generates batches with a median rate of 125 K events per second. Due to the fixed input rate, we assess scaling by looking at the CPU load of each worker.

Figure 11 shows per-machine CPU inverse utilization $1/U$ for $U = \sum_i^N (u_i + s_i) / \sum_i^N t_i$ with user CPU time u_i, system CPU time s_i, and wallclock time t_i, for selected values of i in $[0, N]$. The value U can exceed 1.0 because each node runs several threads, and CPU time measurements yield the sum of all threads. As one would expect for effective load-balancing, we observe linear scaling gains for each added node N.

Query. Our second measurement seeks to understand how query latency changes when varying the number of nodes. We show the index completion time of query D in Figure 12. For these measurements, we first primed the file system cache in each case to compensate for a shortcut that our current implementation takes (it maintains the index in numerous small files that cause high seek penalties for reads from disk; an effect we could avoid by optimizing the disk layout through an intermediary step so that the index can read its data sequentially).

We observe linear scaling from 12 nodes upward, but experience problems for the lower half. Other queries show linear scaling for small numbers of nodes. We are in the process of investigating the discrepancy.

5.4 Storage

Unlike systems which process data in situ, VAST relies on secondary indexes that require additional storage space. In the case of the Bro connection logs, the index increases the total storage by 90%. VAST, however, also compresses the raw data before storing in the archive, in this case cutting it down to 47% of its original size. Taken together, VAST requires 1.37 times the volume of its raw input. For PCAP traces VAST, archives entire packets, but skips all packet payload during index construction. Archive compression brings down the trace to 92% of its original size, whereas the index for connection 4-tuple plus timestamps amounts to 4%. In total, VAST still occupies less space than the original data.

String and container indexes require the most storage, due to their composite and variable-length nature. The remaining indexes exhibit constant space design, and their concrete size is a direct function of encoding and layout of the bit vectors.

6 Conclusion

When security analysts today attempt to reconstruct the sequence of events leading to a cyber incident, they struggle to bring together enormous volumes of heterogeneous data. We present VAST [54], a novel platform for forensic analysis that captures and retains a high-fidelity archive of a network's *entire* activity, leveraging domain-specific semantics to manage high data volumes while supporting rapid queries against historical data. VAST's novelty comes from synthesizing powerful indexing technology with a distributed, entirely asynchronous system architecture that can fully exploit today's highly concurrent architectures. Our evaluation with real-world log and packet data demonstrates the system's potential to support interactive investigation and exploration at a level beyond what current systems offer.

References

[1] Agarwal, R., Khandelwal, A., and Stoica, I. Succinct: Enabling Queries on Compressed Data. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[2] Allman, M., Kreibich, C., Paxson, V., Sommer, R., and Weaver, N. Principles for Developing Comprehensive Network Visibility. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)* (2008).

[3] Antoshenkov, G. Byte-aligned Bitmap Compression. In *Proceedings of the Conference on Data Compression (DCC)* (1995), p. 476.

[4] Armstrong, J. *Making Reliable Distributed Systems in the Presence of Software Errors*. Ph.D. Thesis, Department of Microelectronics and Information Technology, KTH, Sweden, 2003.

[5] Bao, C., Huitema, C., Bagnulo, M., Boucadair, M., and Li, X. IPv6 Addressing of IPv4/IPv6 Translators. RFC 6052, Internet Engineering Task Force (IETF), 2010.

[6] Bayer, R., and McCreight, E. M. Organization and Maintenance of Large Ordered Indexes. In *Record of the ACM SIGFIDET Workshop on Data Description and Access* (1970), pp. 107–141.

[7] Chambi, S., Lemire, D., Kaser, O., and Godin, R. Better Bitmap Performance with Roaring Bitmaps. *CoRR abs/1402.6407* (2014).

[8] Chan, C.-Y., and Ioannidis, Y. E. Bitmap Index Design and Evaluation. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (1998), pp. 355–366.

[9] Chan, C.-Y., and Ioannidis, Y. E. An Efficient Bitmap Encoding Scheme for Selection Queries. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (1999), pp. 215–226.

[10] Charousset, D., Schmidt, T. C., Hiesgen, R., and Wählisch, M. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proceedings of the International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!)* (2013).

[11] Cohen, M. I., Bilby, D., and Caronni, G. Distributed Forensics and Incident Response in the Enterprise. *Digital Investigations 8* (2011), S101–S110.

[12] Colantonio, A., and Di Pietro, R. CONCISE: Compressed 'n' Composable Integer Set. *Information Processing Letters 110*, 16 (2010), 644–650.

[13] Corrales, F., Chiu, D., and Sawin, J. Variable Length Compression for Bitmap Indices. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)* (2011), pp. 381–395.

[14] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the Conference on Symposium on Operating Systems Design & Implementation (OSDI)* (2004), vol. 6, pp. 10–10.

[15] Deliège, F., and Pedersen, T. B. Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (2010), pp. 228–239.

[16] Doherty, W. J., and Thadani, A. J. The Economic Value of Rapid Response Time. *IBM* (1982).

[17] ElasticSearch. `https://www.elastic.co/products/elasticsearch`.

[18] etcd. `https://github.com/coreos/etcd`.

[19] Fusco, F., Dimitropoulos, X., Vlachos, M., and Deri, L. pcapIndex: An Index for Network Packet Traces with Legacy Compatibility. *SIGCOMM Computer Communication Review 42*, 1 (2012), 47–53.

[20] Fusco, F., Stoecklin, M. P., and Vlachos, M. NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 1382–1393.

[21] Fusco, F., Vlachos, M., and Dimitropoulos, X. RasterZip: Compressing Network Monitoring Data with Support for Partial Decompression. In *Proceedings of the Internet Measurement Conference (IMC)* (2012), pp. 51–64.

[22] Giura, P., and Memon, N. NetStore: An Efficient Storage Infrastructure for Network Forensics and Monitoring. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)* (2010), pp. 277–296.

[23] Guzun, G., and Canahuate, G. Performance Evaluation of Word-Aligned Compression Methods for Bitmap Indices. *Knowledge and Information Systems* (2015), 1–28.

[24] GUZUN, G., CANAHUATE, G., CHIU, D., AND SAWIN, J. A Tunable Compression Framework for Bitmap Indices. In *In Proceedings of the International Conference on Data Engineering (ICDE)* (2014), pp. 484–495.

[25] HEWITT, C., BISHOP, P., AND STEIGER, R. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)* (1973), pp. 235–245.

[26] HOARE, C. A. R. Communicating Sequential Processes. *Communications of the ACM 21*, 8 (1978), 666–677.

[27] INGO MLLER, CORNELIUS RATSCH, F. F. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (2014), pp. 283–294.

[28] ISO/IEC. Information technology – Microprocessor Systems – Floating-Point arithmetic. Standard 60559:2011, 2011.

[29] Apache Kafka. `http://kafka.apache.org`.

[30] KNUTH, D. E. *The Art of Computer Programming: Sorting and Searching*, vol. 3. 1998.

[31] KREIBICH, C., WEAVER, N., NECHAEV, B., AND PAXSON, V. Netalyzr: Illuminating the Edge Network. In *Proceedings of the Internet Measurement Conference (IMC)* (2010), pp. 246–259.

[32] LEACH, P. J., MEALLING, M., AND SALZ, R. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, Internet Engineering Task Force (IETF), 2005.

[33] LEE, J., LEE, S., LEE, J., YI, Y., AND PARK, K. FloSIS: A Highly Scalable Network Flow Capture System for Fast Retrieval and Storage Efficiency. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2015).

[34] LEMIRE, D., KASER, O., AND AOUICHE, K. Sorting Improves Word-aligned Bitmap Indexes. *Data & Knowledge Engineering 69*, 1 (2010), 3–28.

[35] libpcap. `http://www.tcpdump.org`.

[36] Lucene. `https://lucene.apache.org`.

[37] LZ4: Extremely Fast Compression algorithm. `https://github.com/Cyan4973/lz4`.

[38] MAIER, G., SOMMER, R., DREGER, H., FELDMANN, A., PAXSON, V., AND SCHNEIDER, F. Enriching Network Security Analysis with Time Travel. In *Proceedings of the ACM SIGCOMM conference* (2008).

[39] MARTNEZ-PRIETO, M. A., BRISABOA, N., CNOVAS, R., CLAUDE, F., AND NAVARRO, G. Practical Compressed String Dictionaries. *Information Systems 56* (2016), 73–108.

[40] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 330–339.

[41] O'NEIL, P. E. Model 204 Architecture and Performance. In *Proceedings of the International Workshop on High Performance Transaction Systems* (1987), pp. 40–59.

[42] ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2014), pp. 305–319.

[43] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks 31*, 23–24 (1999), 2435–2463.

[44] ROTEM, D., STOCKINGER, K., AND WU, K. Optimizing Candidate Check Costs for Bitmap Indices. In *Proceedings of International Conference on Information and Knowledge Management (CIKM)* (2005), pp. 648–655.

[45] SCHMIDT, A., AND BEINE, M. A Concept for a Compression Scheme of Medium-Sparse Bitmaps. In *Proceedings of the International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)* (2011), pp. 192–195.

[46] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST)* (2010), pp. 1–10.

[47] SINHA, R. R., AND WINSLETT, M. Multiresolution Bitmap Indexes for Scientific Data. *ACM Transactions on Database Systems (TODS) 32*, 3 (2007).

[48] Snappy: A fast compressor/decompressor. `https://code.google.com/p/snappy/`.

[49] Splunk. `http://www.splunk.com`.

[50] STOCKINGER, K., CIESLEWICZ, J., WU, K., ROTEM, D., AND SHOSHANI, A. Using bitmap index for joint queries on structured and text data. *New Trends in Data Warehousing and Data Analysis* (2009), 1–23.

[51] TAYLOR, T., COULL, S. E., MONROSE, F., AND MCHUGH, J. Toward Efficient Querying of Compressed Network Payloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2012), USENIX ATC '12.

[52] VALLENTIN, M. *Scalable Network Forensics*. Ph.D. Thesis, University of California, Berkeley, 2016. (*in preparation*).

[53] VAN SCHAIK, S. J., AND DE MOOR, O. A Memory Efficient Reachability Data Structure Through Bit Vector Compression. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2011), pp. 913–924.

[54] VAST. `http://vast.io`.

[55] WONG, H. K. T., LIU, H.-F., OLKEN, F., ROTEM, D., AND WONG, L. Bit Transposed Files. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (1985), pp. 448–457.

[56] WU, K., OTOO, E., AND SHOSHANI, A. On the Performance of Bitmap Indices for High Cardinality Attributes. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (2004), pp. 24–35.

[57] WU, K., OTOO, E. J., AND SHOSHANI, A. A Performance Comparison of Bitmap Indexes. In *Proceedings of International Conference on Information and Knowledge Management (CIKM)* (2001), pp. 559–561.

[58] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012), pp. 2–2.

Appendix

In §4.2.2 we introduce the structure of VAST's high-level index types, but do note elaborating how to operate on them with concrete algorithms. Table 3 and Table 4 provide this information in more detail. For each type, we show how to append a value (symbol ↞) to an index as well as how to query it in terms of logical operations.

The *basic* index I in these tables represents a bitmap or inverted index with a fixed binning, encoding, and compression scheme. I operates on unsigned integer values and supports operators $\{<, \leq, =, \neq, \geq, >\}$. For append and lookup algorithms of the concrete encoding schemes, we refer the reader to the literature [8, 9], from which summarize established results about multi-component indexes in the following.

The basic index forms the foundation for the k-component index $K^\beta = \langle I_1, \ldots, I_k \rangle$ with $|\beta| = k$ (see §4.2.1). It represents the foundation of many higher-level indexes and its lookup algorithm varies according to the relational operator of the predicate. Answering equality queries involves computing a simple conjunction:

$$EQ(i,x) = \bigwedge_{j=1}^{i} (I_j = x_j) \qquad (1)$$

Thus, we can answer $K^\beta = x$ with $EQ(k,x)$, and $K^\beta \neq x$ with $\overline{EQ(k,x)}$. For one-sided range queries, the algorithm *less-than-or-equal* (LE) implements range lookup of the form $K^\beta \leq x$ as follows:

$$LE(i,x) = \begin{cases} (I_i \leq x_i - 1) \vee (\theta_i \wedge LE(i-1,x)) & i > 1, x_i > 0 \\ \theta_i \wedge LE(i-1,x) & i > 1, x_i = 0 \\ (I_i \leq x_i - 1) \vee LE(i-1,x) & i > 1, x_i = \beta_i - 1 \\ I_i \leq x_i & i = 1 \end{cases} \qquad (2)$$

The extra parameter θ_i depends on the coding scheme and means either $I_i = x_i$ or $I_i \leq x_i$. Putting together algorithms EQ and LE, we can now answer $K^\beta \circ x$ under all relational operators with the algorithm ℓ:

$$\ell(K^\beta, \circ, x) = \begin{cases} EQ(k,x) & \circ \in \{=\} \\ \overline{EQ(k,x)} & \circ \in \{\neq\} \\ LE(k,x) & \circ \in \{\leq\} \\ \overline{LE(k,x)} & \circ \in \{>\} \\ LE(k,x-1) & \circ \in \{<\} \wedge x > 0 \\ \overline{LE(k,x-1)} & \circ \in \{\geq\} \wedge x > 0 \\ 0 & \circ \in \{<\} \wedge x = 0 \\ 1 & \circ \in \{\geq\} \wedge x = 0 \end{cases} \qquad (3)$$

The two results 0 and 1 denote the empty and complete identifier set. In the case of bitmap indexes, 0 represents a bit vector with all 0-bits, whereas 1 only consists of 1-bits.

Type	Structure	Append	Lookup
basic*	I	$I \twoheadleftarrow x^{(\alpha)}$	$I \circ x$
k-component†	$K^\beta = \langle I_k, \dots, I_1 \rangle$ $\Theta^k = K^\beta \quad {\scriptstyle \beta_i = \beta_j = 2 \wedge \lvert\beta\rvert = k}$ $\Phi^w = K^\beta \quad {\scriptstyle \Pi_{i=1}^k \beta_i \leq 2^w}$	$K^\beta \twoheadleftarrow x^{(\alpha)} \;\equiv\; I_i \twoheadleftarrow x_i^{(\alpha)} \quad \forall 1 \leq i \leq k$	$K^\beta \circ x \;\equiv\; \ell(K^\beta, \circ, x)$
`bool`	$\mathbb{B} = S$	$\mathbb{B} \twoheadleftarrow x^{(\alpha)} \;\equiv\; S \twoheadleftarrow \alpha \quad \text{iff } x = \texttt{true}$	$\mathbb{B} \circ x \;\equiv\; \begin{cases} \overline{S} & x = \texttt{false} \\ S & x = \texttt{true} \end{cases}$
`count`	$\mathbb{C} = \Phi^{64}$	$\mathbb{C} \twoheadleftarrow x^{(\alpha)} \;\equiv\; \Phi^{64} \twoheadleftarrow x^{(\alpha)}$	$\mathbb{C} \circ x \;\equiv\; \Phi^{64} \circ x$
`int`	$\mathbb{I} = \Phi^{64}$	$\mathbb{I} \twoheadleftarrow x^{(\alpha)} \;\equiv\; \Phi^{64} \twoheadleftarrow (x^{(\alpha)} \overset{u}{+} 2^{63})$	$\mathbb{I} \circ x \;\equiv\; \Phi^{64} \circ (x \overset{u}{+} 2^{63})$
`real`§	$\mathbb{F} = \langle S, E, M \rangle$ $S = \mathbb{B}$ $E = \Theta^{11}$ $M = \Theta^{52}$	$\mathbb{F} \twoheadleftarrow \langle x_s, x_e, x_m \rangle^{(\alpha)} \;\equiv\; \begin{cases} S \twoheadleftarrow x_s^{(\alpha)} \\ E \twoheadleftarrow x_e^{(\alpha)} \\ M \twoheadleftarrow x_m^{(\alpha)} \end{cases}$	$\mathbb{F} \circ x \;\equiv\; \begin{cases} S = x_s \wedge E \circ x_e \wedge M \circ x_m & \circ \in \{=, \neq\} \\ S = 0 \wedge E \circ x_e \wedge M \circ x_m & x \geq 0 \wedge \circ \in \{>, \geq\} \\ S = 1 \vee (E \circ x_e \wedge M \circ x_m) & x \geq 0 \wedge \circ \in \{<, \leq\} \\ S = 0 \vee (E \triangledown x_e \wedge M \triangledown x_m) & x < 0 \wedge \circ \in \{>, \geq\} \\ S = 1 \wedge E \triangledown x_e \wedge M \triangledown x_m & x < 0 \wedge \circ \in \{<, \leq\} \end{cases}$
`duration`	$\mathbb{D} = \mathbb{I}$	$\mathbb{D} \twoheadleftarrow x^{(\alpha)} \;\equiv\; \mathbb{I} \twoheadleftarrow x^{(\alpha)}$	$\mathbb{D} \circ x \;\equiv\; \mathbb{I} \circ x$
`time`	$\mathbb{T} = \mathbb{D}$	$\mathbb{T} \twoheadleftarrow x^{(\alpha)} \;\equiv\; \mathbb{D} \twoheadleftarrow x^{(\alpha)}$	$\mathbb{T} \circ x \;\equiv\; \mathbb{D} \circ x$
`string`	$\mathbb{S} = \langle \phi, \kappa_1, \dots, \kappa_M \rangle$ $\phi = K^\beta$ $\kappa_i = \Theta^8$	$\mathbb{S} \twoheadleftarrow \langle x_1, \dots, x_n \rangle^{(\alpha)} \;\equiv\; \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ \kappa_i \twoheadleftarrow x_i^{(\alpha)} \quad \forall 1 \leq i \leq n \end{cases}$	$\mathbb{S} \circ x \;\equiv\; \begin{cases} \mathbf{0} & \lvert x\rvert > M \\ \phi = 0 & \lvert x\rvert = 0 \\ \phi = \lvert x\rvert \wedge \bigwedge_{i=1}^{\lvert x\rvert} \kappa_i = x_i & \circ \in \{=, \neq\} \\ \phi \geq \lvert x\rvert \wedge \bigvee_{i=1}^{M-\lvert x\rvert+1} \left(\bigwedge_{j=1}^{\lvert x\rvert} \kappa_{i+j-1} = x_j \right) & \circ \in \{\in, \notin\} \end{cases}$

[*] The *basic* index has a fixed binning, coding, and compression scheme and operates on values $x \in X \subseteq \mathbb{N}_0^+$ and has cardinality $C \leq |X|$. (see §4.2.1)

[†] The k-component index operates with a base $\beta = \langle \beta_k, \dots, \beta_1 \rangle$. We show algorithm ℓ in §**??**. The bit-sliced index [55] is a special of K^β where $\beta_i = \beta_i = 2$ for all $i \neq j$. The multi-component index Φ^w can at most represent 2^w distinct values.

[§] We denote by $\triangledown$ the "mirrored" operator of $\circ$, e.g., $<$ and $>$.

Table 3: Summary of append and lookup operations on high-level indexes.

Type	Structure	Append	Lookup
`addr`	$\mathbb{A} = \Theta^{128}$	$\mathbb{A} \twoheadleftarrow \langle x_1, \ldots, x_{128} \rangle^{(\alpha)} \equiv \Theta_i^{128} \leftarrow x_i^{(\alpha)} \quad \forall 1 \le i \le 128$	$\mathbb{A} \circ x \equiv \begin{cases} \bigwedge_{i=1}^{k} \Theta_i^{128} = x_i & \circ \in \{\in\} \\ \overline{\mathbb{A} \ni x} & \circ \in \{\notin\} \end{cases}$
`subnet`	$\mathbb{U} = \langle \mathbb{A}, \Phi^8 \rangle$	$\mathbb{U} \twoheadleftarrow \langle x_a, x_p \rangle^{(\alpha)} \equiv \begin{cases} \mathbb{A} \twoheadleftarrow x_a^{(\alpha)} \\ \Phi^8 \twoheadleftarrow x_p^{(\alpha)} \end{cases}$	$\mathbb{U} \circ x \equiv \begin{cases} \Phi^8 \le x_p \wedge \left(\bigwedge_{i=1}^{p} \mathbb{A}_i = x_{a_i} \right) & \circ \in \{\in\} \\ \overline{\mathbb{U} \ni x} & \circ \in \{\notin\} \end{cases}$
`port`	$\mathbb{P} = \langle \Phi^{16}, \Phi^2 \rangle$	$\mathbb{P} \twoheadleftarrow \langle x_n, x_t \rangle^{(\alpha)} \equiv \begin{cases} \Phi^{16} \twoheadleftarrow x_n^{(\alpha)} \\ \Phi^2 \twoheadleftarrow x_t^{(\alpha)} \end{cases}$	$\mathbb{P} \circ x \equiv \begin{cases} \Phi^{16} \circ x_n \wedge \Phi^2 = x_t & x_t \neq \texttt{unknown} \\ \Phi^{16} \circ x_n & x_t = \texttt{unknown} \end{cases}$
`vector`[*]	$\mathbb{X}^V = \langle \phi, \mathbb{V}_1, \ldots, \mathbb{V}_M \rangle$ $\phi = K^\beta$	$\mathbb{X}^V \twoheadleftarrow \langle x_1, \ldots, x_n \rangle^{(\alpha)} \equiv \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ \mathbb{V}_i \twoheadleftarrow x_i^{(\alpha)} & \forall 1 \le i \le n \end{cases}$	$\mathbb{X}^V \circ x \equiv \begin{cases} \text{see } \mathbb{S} \circ x & \tau(x) = \texttt{vector} \\ \text{see } \mathbb{X}^S \circ x & \tau(x) = \texttt{set} \end{cases}$
`set`	$\mathbb{X}^S = \langle \phi, \mathbb{V}_1, \ldots, \mathbb{V}_M \rangle$ $\phi = K^\beta$	$\mathbb{X}^S \twoheadleftarrow \langle x_1, \ldots, x_n \rangle^{(\alpha)} \equiv \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ \mathbb{V}_i \twoheadleftarrow x_i^{(\alpha)} & \forall 1 \le i \le n \end{cases}$	$\mathbb{X}^S \circ x \equiv \begin{cases} \mathrm{o} & \lvert x \rvert > M \\ \phi = 0 & \lvert x \rvert = 0 \\ \bigwedge_{i=1}^{\lvert x \rvert} \left(\bigvee_{j=1}^{M} \mathbb{V}_j = x_i \right) & \textit{otherwise} \end{cases}$
`table`[†]	$\mathbb{X}^T = \langle \phi, X_1, \ldots, X_M, Y_1, \ldots, Y_M \rangle$ $\phi = K^\beta$ $X_i = \mathbb{V}$ $Y_i = \mathbb{V}$	$\mathbb{X}^T \twoheadleftarrow \langle (k_1, v_1) \ldots, (k_n, v_n) \rangle^{(\alpha)} \equiv \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ X_i \twoheadleftarrow k_i^{(\alpha)} & \forall 1 \le i \le n \\ Y_i \twoheadleftarrow v_i^{(\alpha)} & \forall 1 \le i \le n \end{cases}$	$\mathbb{X}^T \circ k \equiv \begin{cases} \bigvee_{i=1}^{M} X_i = k & \circ \in \{\in\} \\ \overline{\mathbb{X}^T \ni k} & \circ \in \{\notin\} \end{cases}$ $\mathbb{X}^T \circ v \equiv \begin{cases} \bigvee_{i=1}^{M} Y_i = v & \circ \in \{\in\} \\ \overline{\mathbb{X}^T \ni v} & \circ \in \{\notin\} \end{cases}$ $\mathbb{X}^T \circ (k, v) \equiv \begin{cases} \bigvee_{i=1}^{M} (X_i = k \wedge Y_i = v) & \circ \in \{\in\} \\ \overline{\mathbb{X}^T \ni (k \to v)} & \circ \in \{\notin\} \end{cases}$

[*] Depending on the type $\tau(x)$ of value x, the lookup function can either preserve ordering (as in substring search) or ignore ordering (as in subset search).

[†] A table value has the form $x = \langle (k_1, v_1), \ldots, (k_n, v_n) \rangle$. We show lookups for a single key, value, or mapping.

Table 4: Summary of append and lookup operations on high-level indexes.

Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics

Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, Ion Stoica
University of California, Berkeley

Abstract

Recent workload trends indicate rapid growth in the deployment of machine learning, genomics and scientific workloads on cloud computing infrastructure. However, efficiently running these applications on shared infrastructure is challenging and we find that choosing the right hardware configuration can significantly improve performance and cost. The key to address the above challenge is having the ability to predict performance of applications under various resource configurations so that we can automatically choose the optimal configuration.

Our insight is that a number of jobs have predictable structure in terms of computation and communication. Thus we can build performance models based on the behavior of the job on small samples of data and then predict its performance on larger datasets and cluster sizes. To minimize the time and resources spent in building a model, we use optimal experiment design, a statistical technique that allows us to collect as few training points as required. We have built Ernest, a performance prediction framework for large scale analytics and our evaluation on Amazon EC2 using several workloads shows that our prediction error is low while having a training overhead of less than 5% for long-running jobs.

1 Introduction

In the past decade we have seen a rapid growth of large-scale *advanced* analytics that implement complex algorithms in areas like distributed natural language processing [24, 74], deep learning for image recognition [34], genome analysis [72, 61], astronomy [17] and particle accelerator data processing [19]. These applications differ from traditional analytics workloads (e.g., SQL queries) in that they are not only data-intensive but also computation-intensive, and typically run for a long time (and hence are expensive). Along with new workloads, we have seen widespread adoption of cloud computing with large data sets being hosted [7, 1], and the emergence of sophisticated analytics services, such as machine learning, being offered by cloud providers [9, 6].

With cloud computing environments such as Amazon EC2, users typically have a large number of choices in terms of the instance types and number of instances they can run their jobs on. Not surprisingly, the amount of memory per core, storage media, and the number of instances are crucial choices that determine the running time and thus indirectly the cost of running a given job. Using common machine learning kernels we show in §2.2 that choosing the right configuration can improve performance by up to 1.9x at the same cost.

In this paper, we address the challenge of choosing the configuration to run large advanced analytics applications in heterogeneous multi-tenant environments. The choice of configuration depends on the user's goals which typically includes either minimizing the running time given a budget or meeting a deadline while minimizing the cost. The key to address this challenge is developing a performance prediction framework that can accurately predict the running time on a specified hardware configuration, given a job and its input.

One approach to address this challenge is to predict the performance of a job based on monitoring the job's previous runs [39, 44]. While simple, this approach assumes the job runs repeatedly on the same or "similar" data sets. However, this assumption does not always hold. First, even when a job runs periodically it typically runs on data sets that can be widely different in both size and content. For example, a prediction algorithm may run on data sets corresponding to different days or time granularities. Second, workloads such as interactive machine learning [9, 55] and parameter tuning generate unique jobs for which we have little or no relevant history. Another approach to predict job performance is to build a detailed parametric model for the job. Along these lines, several techniques have been recently proposed in the context of MapReduce-like frameworks [77, 52]. These techniques have been aided by the inherent simplicity of the two-stage MapReduce model. However, the recent increase in the popularity of more complex parallel computation engines such as Dryad [51] and Spark [83] make these parametric techniques much more difficult to apply.

In this paper, we propose a new approach that can accurately predict the performance of a given analytics job. The main idea is to run a set of instances of the entire job on samples of the input, and use the data from these training runs to create a performance model. This approach has low overhead, as in general it takes much less time and resources to run the training jobs than running the job itself. Despite the fact that this is a black-box approach (i.e., requires no knowledge about the internals of

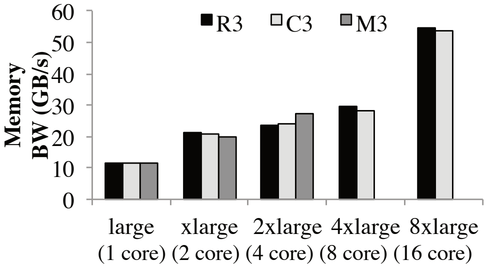


Figure 1: **Comparison of memory bandwidths across Amazon EC2 `m3`/`c3`/`r3` instance types. There are only three sizes for `m3`. Smaller instances (large, xlarge) have better memory bandwidth per core.**

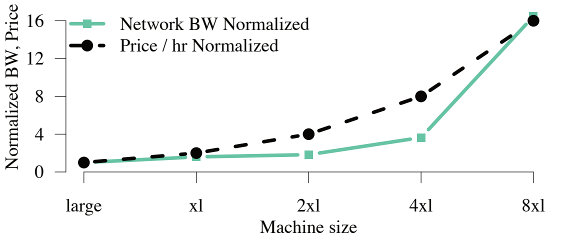


Figure 2: **Comparison of network bandwidths with prices across different EC2 `r3` instance sizes normalized to `r3.large`. `r3.8xlarge` has the highest bandwidth per core.**

the job), it works surprisingly well in practice.

The reason this approach works so well is because many advanced analytics workloads have a simple structure and the dependence between their running times and the input sizes or number of nodes is in general characterized by a relatively small number of smooth functions. This should come as no surprise as big data has naturally lead researchers and practitioners to develop algorithms that are linear [22] or quasi-linear in terms of the input size, and which scale well with the number of nodes. As a simple example, consider a mini-batch gradient descent algorithm used for linear regression. For a dataset with *m* data points and *n* features per partition, the time taken by each task to compute the gradient is uniform (*mn*) and similarly the size of output from every task is the same, a vector of size *n*.

The cost and utility of training data points collected is important for low-overhead prediction and we address this problem using *optimal experiment design* [63] , a statistical technique that allows us to select the most useful data points for training. We augment experiment design with a cost model and this helps us find the training data points to explore within a given budget. We have built support for the above techniques in Ernest and we find that a number of advanced analytics workloads can be accurately modeled using simple features that reflect commonly found computation and communication patterns. We include a cross-validation based verification scheme in Ernest to detect when a workload does not match the features being used and show how we can easily extend our model in such cases.

Using Amazon EC2 as our execution environment, we evaluate the accuracy of our system using a number of workloads including (a) several machine learning algorithms that are part of Spark MLlib [56], (b) queries from GenBase [73] and I/O intensive transformations using ADAM [61] on a full genome, and (c) a speech recognition pipeline that achieves state-of-the-art results [50]. Our evaluation shows that our average prediction error is under 20% and that this is sufficient for choosing the appropriate number or type of instances. Our training overhead for long-running jobs is less than 5% and we also find that using experiment design improves prediction error for some algorithms by $30-50\%$ over a cost-based scheme. Finally, using our predictions we show that for a long-running speech recognition pipeline, finding the appropriate number of instances can reduce cost by around $4x$ compared to a greedy allocation scheme. In summary, the main contributions of this paper are:

- We propose Ernest, a performance prediction framework that works with unmodified jobs and achieves low overhead using optimal experiment design.
- We show how Ernest can detect when a model isn’t appropriate and how small extensions can be used to model complex workloads.
- Using experiments on EC2, we show that Ernest is accurate for a variety of algorithms, input sizes, and cluster sizes.

2 Background

In this section we first present an overview of different approaches to performance prediction. We then discuss recent hardware and workload trends for large scale data analysis. We also present an example of an end-to-end machine learning pipeline and discuss some of the computation and communication patterns that we see using this example.

2.1 Performance Prediction

Performance modeling and prediction have been used in many different contexts in various systems [59, 16, 39]. At a high level performance modeling and prediction proceeds as follows: select an output or response variable that needs to be predicted and the features to be used for prediction. Next, choose a relationship or a model that can provide a prediction for the output variable given the input features. This model could be rule based [27, 21] or use machine learning techniques [60, 80] that build an estimator using some training data. We focus on machine learning based techniques in this paper and we next discuss two major approaches in modeling that influences the training data and machine learning algorithms used.

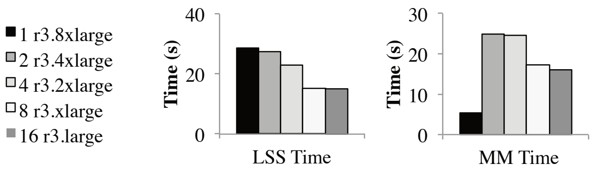
Performance counters: Performance counter based approaches typically use a large number of low level counters to try and predict application performance characteristics. Such an approach has been used with CPU counter for profiling [14], performance diagnosis [81, 25] and virtual machine allocation [60]. A similar approach has also been used for analytics jobs where the MapReduce counters have been used for performance prediction [77] and straggler mitigation [80]. Performance-counter based approaches typically use advanced learning algorithms like random forests, SVMs. However as they use a large number of features, they require large amounts of training data and are well suited for scenarios where historical data is available.

System modeling: In the system modeling approach, a performance model is developed based on the properties of the system being studied. This method has been used in scientific computing [16] for compilers [11], programming models [21, 27]; and by databases [29, 57] for estimating the progress made by SQL queries. System design based models are usually simple and interpretable but may not capture all the execution scenarios. However one advantage of this approach is that only a small amount of training data is required to make predictions.

In this paper, we look at how to perform efficient performance prediction for large scale advanced analytics. We use a system modeling approach where we build a high-level end-to-end model for advanced analytics jobs. As collecting training data can be expensive, we further focus on how to minimize the amount of training data required in this setting. We next survey recent hardware and workload trends that motivate this problem.

2.2 Hardware Trends

The widespread adoption of cloud computing has led to a large number of data analysis jobs being run on cloud computing platforms like Amazon EC2, Microsoft Azure and Google Compute Engine. In fact, a recent survey by Typesafe of around 500 enterprises [4] shows that 53% of Apache Spark users deploy their code on Amazon EC2. However using cloud computing instances comes with its own set of challenges. As cloud computing providers use virtual machines for isolation between users, there are a number of fixed-size virtual machine options that users can choose from. Instance types vary not only in capacity (i.e. memory size, number of cores etc.) but also in performance. For example, we measured memory bandwidth and network bandwidth across a number of instance types on Amazon EC2. From Figure 1 we can see that the smaller instances i.e. `large` or `xlarge` have the highest memory bandwidth available per core while Figure 2 shows that `8xlarge` instances have the highest network bandwidth available per core. Based on our experiences with Amazon EC2, we believe these performance variations are not necessarily due to poor isolation between tenants but are instead related to how various instance types are mapped to shared physical hardware.



(a) (b)

Figure 3: **Performance comparison of a Least Squares Solver (LSS) job and Matrix Multiply (MM) across similar capacity configurations.**

The non-linear relationship between price vs. performance is not only reflected in micro-benchmarks but can also have a significant effect on end-to-end performance. For example, we use two machine learning kernels: (a) A least squares solver used in convex optimization [37] and (b) a matrix multiply operation [75], and measure their performance for similar capacity configurations across a number of instance types. The results (Figure 3(a)) show that picking the right instance type can improve performance by up to 1.9x at the same cost for the least squares solver. Earlier studies [47, 79] have also reported such performance variations for other applications like SQL queries, key-value stores. These performance variations motivate the need for a performance prediction framework that can automate the choice of hardware for a given computation.

Finally, performance prediction is important not just in cloud computing but it is also useful in other shared computing scenarios like private clusters. Cluster schedulers [15] typically try to maximize utilization by packing many jobs on a single machine and predicting the amount of memory or number of CPU cores required for a computation can improve utilization [36]. Next, we look at workload trends in large scale data analysis and how we can exploit workload characteristics for performance prediction.

2.3 Workload trends

The last few years have seen the growth of advanced analytics workloads like machine learning, graph processing and scientific analyses on large datasets. Advanced analytics workloads are commonly implemented on top of data processing frameworks like Hadoop [35], Naiad [58] or Spark [83] and a number of high level libraries for machine learning [56, 2] have been developed on top of these frameworks. A survey [4] of Apache Spark users shows that around 59% of them use the machine learning library in Spark and recently launched services like Azure ML [9] provide high level APIs which implement commonly used machine learning algorithms.

Advanced analytics workloads differ from other workloads like SQL queries or stream processing in a number of ways. These workloads are typically numerically

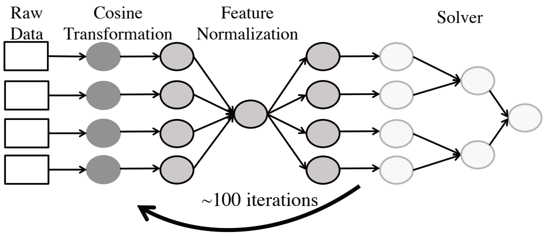


Figure 4: **Execution DAG of a machine learning pipeline used for speech recognition [50]. The pipeline consists of featurization and model building steps which are repeated for many iterations.**

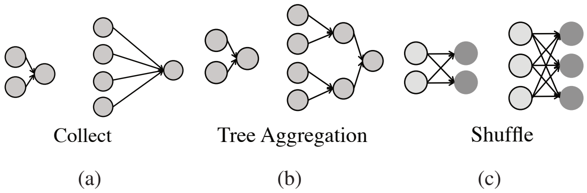


Figure 5: **Scaling behaviors of commonly found communication patterns as we increase the number of machines.**

intensive, i.e. performing floating point operations like matrix-vector multiplication or convolutions [32], and thus are sensitive to the number of cores and memory bandwidth available. Further, such workloads are also often iterative and repeatedly perform parallel operations on data cached in memory across a cluster. Advanced analytics jobs can also be long-running: for example, to obtain the state-of-the-art accuracy on tasks like image recognition [34] and speech recognition [50], jobs are run for many hours or days.

Example: Speech Recognition Pipeline. As an example of an advanced analytics job, we consider a speech recognition pipeline [50] that achieves state-of-the-art accuracy on the TIMIT [26] dataset. The pipeline trains a model using kernel SVMs and the execution DAG is shown in Figure 4. From the figure we can see that such pipelines consist of a number of stages, each of which may be repeated for some iterations. This TIMIT pipeline contains three main stages. The first stage of the DAG reads input data, and featurizes the data by applying a random cosine transformation [64] to each record. Assuming dense input data and equal-sized partitions, we can also see that each task in the first stage will take a similar amount of time to compute. Further, we observe that, unlike SQL queries with selectivity clauses, the transformation here results in the same amount of output data per input record across all tasks. The second stage in the pipeline normalizes data, which requires computing the mean and variance of the features by aggregating values across all the partitions. In the last stage, the normalized features are fed into a convex solver [23] to build a model. The model is then refined by generating more features and these steps are repeated for 100 iterations to achieve state-of-the-art accuracy.

Workload Properties: Since advanced analytics jobs run on large datasets are expensive, we observe that developers have focused on algorithms that are scalable across machines and are of low complexity (e.g., linear or quasi-linear) [22]. Otherwise, using these algorithms to process huge amounts of data might be infeasible. The natural outcome of these efforts is that these workloads admit relatively simple performance models. Specifically, we find that the computation required per data item remains the same as we scale the computation.

Further, we observe that only a few communication patterns repeatedly occur in such jobs. These patterns (Figure 5) include (a) the all-to-one or *collect* pattern, where data from all the partitions is sent to one machine, (b) *tree-aggregation* pattern where data is aggregated using a tree-like structure, and (c) a *shuffle* pattern where data goes from many source machines to many destinations. These patterns are not specific to advanced analytics jobs and have been studied before [30, 20]. Having a handful of such patterns means that we can try to automatically infer how the communication costs change as we increase the scale of computation. For example, assuming that data grows as we add more machines (i.e., the data per machine is constant), the time taken for the collect increases as $O(machines)$ as a single machine needs to receive all the data. Similarly the time taken for a binary aggregation tree grows as $O(log(machines))$.

Finally we observe that many algorithms are iterative in nature and that we can also *sample the computation* by running just a few iterations of the algorithm. Next we will look at the design of the performance model.

3 Modeling Advanced Analytics Jobs

In this section we outline a model for predicting execution time of advanced analytics jobs. This scheme only uses end-to-end running times collected from executing the job on smaller samples of the input and we discuss techniques for model building and data collection.

At a high level we consider a scenario where a user provides as input a parallel job (written using any existing data processing framework) and a pointer to the input data for the job. We do not assume the presence of any historical logs about the job and our goal here is to build a model that will predict the execution time for any input size, number of machines for this given job. The main steps in building a predictive model are (a) determining what training data points to collect (b) determining what features should be derived from the training data and (c) performing feature selection to pick the simplest model that best fits the data. We discuss all three aspects below.

3.1 Features for Prediction

One of the consequences of modeling end-to-end unmodified jobs is that there are only a few parameters that we can change to observe changes in performance. Assuming that the job, the dataset and the machine types are fixed, the two main features that we have are (a) the number of rows or fraction of data used (scale) and (b) the number of machines used for execution. Our goal in the modeling process is to derive as few features as required for the amount of training data required grows linearly with the number of features.

To build our model we add terms related to the computation and communication patterns discussed in §2.3. The terms we add to our linear model are (a) a fixed cost term which represents the amount of time spent in serial computation (b) the interaction between the scale and the inverse of the number of machines; this is to capture the *parallel* computation time for algorithms whose computation scales *linearly* with data, i.e., if we double the size of the data with the same number of machines, the computation time will grow linearly (c) a $log(machines)$ term to model communication patterns like aggregation trees (d) a linear term $O(machines)$ which captures the all-to-one communication pattern and fixed overheads like scheduling / serializing tasks (i.e. overheads that scale as we add more machines to the system). Note that as we use a linear combination of *non-linear features*, we can model non-linear behavior as well.

Thus the overall model we are fitting tries to learn values for $\theta_0, \theta_1, \theta_2$, and θ_3 in the formula

$$\begin{aligned} time = \theta_0 + \theta_1 \times (scale \times \frac{1}{machines}) + \\ \theta_2 \times \log(machines) + \\ \theta_3 \times machines \end{aligned} \tag{1}$$

Given these features, we then use a *non-negative least squares* (NNLS) solver to find the model that best fits the training data. NNLS fits our use case very well as it ensures that each term contributes some non-negative amount to the overall time taken. This avoids over-fitting and also avoids corner cases where say the running time could become negative as we increase the number of machines. NNLS is also useful for feature selection as it sets coefficients which are not relevant to a particular job to zero. For example, we trained a NNLS model using 7 data points on all of the machine learning algorithms that are a part of MLlib in Apache Spark 1.2. The final model parameters are shown in Table 1. From the table we can see two main characteristics: (a) that not all features are used by every algorithm and (b) that the contribution of each term differs for each algorithm. These results also show why we cannot reuse models across jobs.

Additional Features: While the features used above capture most of the patterns that we see in jobs, there could other patterns which are not covered. For example in linear algebra operators like QR decomposition the computation time will grow as $scale^2/machines$ if we scale the number of columns. We discuss techniques to detect when the model needs such additional terms in §3.4.

Benchmark	$intercept$	$scale/mc$	mc	$log(mc)$
spearman	0.00	4887.10	0.00	4.14
classification	0.80	211.18	0.01	0.90
pca	6.86	208.44	0.02	0.00
naive.bayes	0.00	307.48	0.00	1.00
summary stats	0.42	39.02	0.00	0.07
regression	0.64	630.93	0.09	1.50
als	28.62	3361.89	0.00	0.00
kmeans	0.00	149.58	0.05	0.54

Table 1: **Models built by Non-Negative Least Squares for MLlib algorithms using `r3.xlarge` instances. Not all features are used by every algorithm.**

3.2 Data collection

The next step is to collect training data points for building a predictive model. For this we use the input data provided by the user and run the *complete* job on small samples of the data and collect the time taken for the job to execute. For iterative jobs we allow Ernest to be configured to run a certain number of iterations (§4). As we are not concerned with the accuracy of the computation we just use the first few rows of the input data to get appropriately sized inputs.

How much training data do we need?: One of the main challenges in predictive modeling is minimizing the time spent on collecting training data while achieving good enough accuracy. As with most machine learning tasks, collecting more data points will help us build a better model but there is time and a cost associated with collecting training data. As an example, consider the model shown in Table 1 for *kmeans*. To train this model we used 7 data points and we look at the importance of collecting additional data by comparing two schemes: in the first scheme we collect data in an increasing order of machines and in the second scheme we use a mixed strategy as shown in Figure 6. From the figure we make two important observations: (a) in this case, the mixed strategy gets to a lower error quickly; after three data points we get to less than 15% error. (b) We see a trend of diminishing returns where adding more data points does not improve accuracy by much. We next look at techniques that will help us find how much training data is required and what those data points should be.

3.3 Optimal Experiment Design

To improve the time taken for training without sacrificing the prediction accuracy, we outline a scheme based on *optimal experiment design*, a statistical technique that can be used to minimize the number of experiment runs required. In statistics, experiment design [63] refers to the study of how to collect data required for any experiment given the modeling task at hand. *Optimal* experiment design specifically looks at how to choose experiments that are optimal with respect to some statistical criterion. At a high-level the goal of experiment design is to determine data points that can give us most information to build an accurate model.

More formally, consider a problem where we are trying to fit a linear model X given measurements $y_1, \ldots, y_m$ and features $a_1, \ldots, a_m$ for each measurement. Each feature vector could in turn consist of a number of dimensions (say n dimensions). In the case of a linear model we typically estimate X using linear regression. We denote this estimate as $\hat{X}$ and $\hat{X} - X$ is the estimation error or a measure of how far our model is from the true model.

To measure estimation error we can compute the Mean Squared Error (MSE) which takes into account both the bias and the variance of the estimator. In the case of the linear model above if we have m data points each having n features, then the variance of the estimator is represented by the $n \times n$ covariance matrix $(\sum_{i=1}^{m} a_i a_i^T)^{-1}$. The key point to note here is that the covariance matrix only depends on the feature vectors that were used for this experiment and not on the model that we are estimating.

In optimal experiment design we choose feature vectors (i.e. a_i) that minimize the estimation error. Thus we can frame this as an optimization problem where we minimize the estimation error subject to constraints on the number of experiments. More formally we can set λ_i as the fraction of times an experiment is chosen and minimize the trace of the inverse of the covariance matrix:

$$\begin{aligned} \text{Minimize} \quad & \mathbf{tr}((\sum_{i=1}^{m} \lambda_i a_i a_i^T)^{-1}) \\ \text{subject to} \quad & \lambda_i \geq 0, \lambda_i \leq 1 \end{aligned}$$

Using Experiment Design: The predictive model described in the previous section can be formulated as an experiment design problem. Given bounds for the scale and number of machines we want to explore, we can come up with all the features that can be used. For example if the scale bounds range from say 1% to 10% of the data and the number of machine we can use ranges from 1 to 5, we can enumerate 50 different feature vectors from all the scale and machine values possible. We can then feed these feature vectors into the experiment design setup described above and only choose to run those experiments whose λ values are non-zero.

Accounting for Cost: One additional factor we need to consider in using experiment design is that each experiment we run costs a different amount. This cost could be in terms of time (i.e. it is more expensive to train with larger fraction of the input) or in terms of machines (i.e. there is a fixed cost to say launching a machine). To account for the cost of an experiment we can augment the optimization problem we setup above with an additional constraint that the total cost should be lesser than some budget. That is if we have a cost function which gives us a cost c_i for an experiment with scale s_i and m_i machines, we add a constraint to our solver that $\sum_{i=1}^{m} c_i \lambda_i \leq B$ where B is the total budget. For the rest of this paper we use the time taken to collect training data as the cost and ignore any machine setup costs as we usually amortize that over all the data we need to collect. However we can plug-in in any user-defined cost function in our framework.

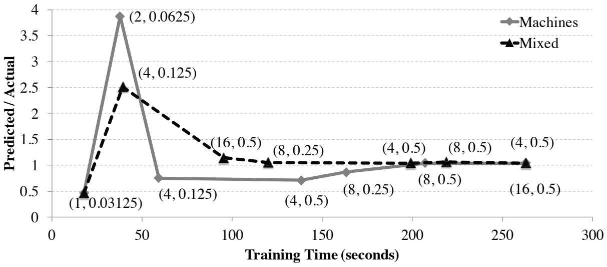


Figure 6: **Comparison of different strategies used to collect training data points for KMeans. The labels next to the data points show the (number of machines, scale factor) used.**

	Residual Sum	Percentage Err	
	of Squares	Median	Max
without $\sqrt{n}$	1409.11	12.2%	64.9%
with $\sqrt{n}$	463.32	5.7%	26.5%

Table 2: **Cross validation metrics comparing different models for Sparse GLM run on the splice-site dataset.**

3.4 Model extensions

The model outlined in the previous section accounts for the most common patterns we see in advanced analytics applications. However there are some complex applications like randomized linear algebra [43] which might not fit this model. For such scenarios we discuss two steps: the first is adding support in Ernest to detect when the model is not adequate and the second is to easily allow users to extend the model being used.

Cross-Validation: The most common technique for testing if a model is valid is to use hypothesis testing and compute test statistics (e.g., using the t-test or the chi-squared test) and confirm the null hypothesis that data belongs to the distribution that the model describes. However as we use non-negative least squares (NNLS) the residual errors are not normally distributed and simple techniques for computing confidence limits, p-values are not applicable. Thus we use *cross-validation*, where subsets of the training data can be used to check if the model will generalize well. There are a number of methods to do cross-validation and as our training data size is small, we use a leave-one-out-cross-validation scheme in Ernest. Specifically if we have collected m training data points, we perform m cross-validation runs where each run uses $m-1$ points as training data and tests the model on the left out data point.

Model extension example: As an example, we consider the GLM classification implementation in Spark MLLib for sparse datasets. In this workload the computation is linear but the aggregation uses two stages (instead of an aggregation tree) where the first aggregation stage has $\sqrt{n}$ tasks for n partitions of data and the second aggregation stage combines the output of $\sqrt{n}$ tasks using one task. This communication pattern is not captured in our model from earlier and the results from cross validation using our original model are shown in Table 2. As we can see in the table both the residual sum of squares and the percentage error in prediction are high for the original model. Extending the model in Ernest with additional terms is simple and in this case we can see that adding the $\sqrt{n}$ term makes the model fit much better. In practice we use a configurable threshold on the percentage error to determine if the model fit is poor. We investigate the end-to-end effects of using a better model in §6.6.

4 Implementation

Ernest is implemented using Python as multiple modules. The modules include a job submission tool that submits training jobs, a training data selection process which implements experiment design using a CVX solver [42, 41] and finally a model builder that uses NNLS from SciPy [53]. Even for a large range of scale and machine values we find that building a model takes only a few seconds and does not add any overhead. In the rest of this section we discuss the job submission tool and how we handle sparse datasets, stragglers.

4.1 Job Submission Tool

Ernest extends existing job submission API [5] that is present in Apache Spark 1.2. This job submission API is similar to Hadoop's Job API [10] and similar job submission APIs exist for dedicated clusters [65, 78] as well. The job submission API already takes in the binary that needs to run (a JAR file in the case of Spark) and the input specification required for collecting training data.

We add a number of optional parameters which can be used to configure Ernest. Users can configure the minimum and maximum dataset size that will be used for training. Similarly the maximum number of machines to be used for training can also be configured. Our prototype implementation of Ernest uses Amazon EC2 and we amortize cluster launch overheads across multiple training runs i.e., if we want to train using 1, 2, 4 and 8 machines, we launch a 8 machine cluster and then run all of these training jobs in parallel.

The model built using Ernest can be used in a number of ways. In this paper we focus on a cloud computing use case where we can choose the number and type of EC2 instances to use for a given application. To do this we build one model per instance type and explore different sized instances (i.e. r3.large,...r3.8xlarge). After training the models we can answer higher level questions like selecting the cheapest configuration given a time bound or picking the fastest configuration given a budget. One of the challenges in translating the performance prediction into a higher-level decision is that the predictions could have some error associated with them. To help with this, we provide the cross validation results (§3.4) along with the prediction and these can be used to compute the range of errors observed on training data. Additionally we plan to provide support for visualizing the scaling behavior and Figure 20 in §6.6 shows an example.

4.2 Handling Sparse Datasets

One of the challenges in Ernest is to deal with algorithms that process sparse datasets. Because of the difference in sparsity across data items, each record could take different time to process. We observe that operations on sparse datasets depend on the number of non-zero entries and thus if we can sample the data such that we use a *representative* sparse subset during training, we should be able to apply modeling techniques described before. However in practice, we don't see this problem as even if there is a huge skew in sparsity across rows, the skew across partitions is typically smaller.

To illustrate, we chose three of the largest sparse datasets that are part of the LibSVM repository [70, 82] and we measured the maximum number of non-zero entries present in every partition after loading the data into HDFS. We normalize these values across partitions and a CDF of partition densities is shown in Figure 7. We observe the the difference in sparsity between the most loaded partition and the least loaded one is less than 35% for all datasets and thus picking a random sample of partitions [76] is sufficient to model computation costs.

4.3 Straggler mitigation by over-allocation

The problem of dealing with stragglers, or tasks which take much longer than other tasks is one of the main challenges in large scale data analytics [80, 13, 33]. Using cloud computing instances could further aggravate the problem due to differences in performance across instances. One technique that we use in Ernest to overcome variation among instances is to launch a small percentage of extra instances and then discard the worst performing among them before running the user's job. We use memory bandwidth and network bandwidth measurements (§2) to determine the slowest instances.

In our experiences with Amazon EC2 we find that even having a few extra instances can be more than sufficient in eliminating the slowest machines. To demonstrate this, we set the target cluster size as $N = 50$ `r3.2xlarge` instances and have Ernest automatically allocate a small percentage of extra nodes. We then run

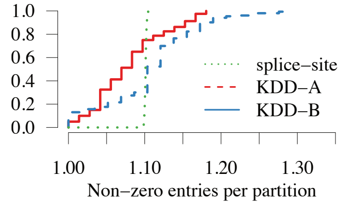


Figure 7: **CDF of maximum number of non-zero entries in a partition, normalized to the least loaded partition for sparse datasets.**

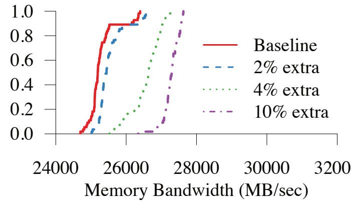


Figure 8: **CDFs of STREAM memory bandwidths under four allocation strategies. Using a small percentage of extra instances removes stragglers.**

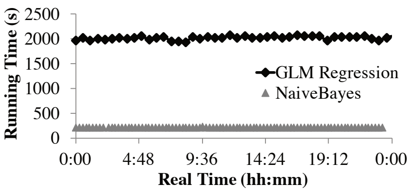


Figure 9: **Running times of GLM and Naive Bayes over a 24-hour time window on a 64-node EC2 cluster.**

STREAM [54] at 30 second intervals and collect memory bandwidth measurements on all instances. Based on the memory bandwidths observed, we eliminate the slowest nodes from the cluster. Figure 8 shows for each allocation strategy, the CDF of the memory bandwidth obtained when picking the best *N* instances from all the instances allocated. We see that Ernest only needs to allocate as few as 2 (or 4%) extra instances to eliminate the slowest stragglers and improve the target cluster's average memory bandwidth from 24.7 GB/s to 26 GB/s.

5 Discussion

In this section we look at *when* a model should be retrained and also discuss the trade-offs associated with including more fine-grained information in Ernest.

5.1 Model reuse

The model we build using Ernest predicts the performance for a given job for a specific dataset and a target cluster. One of the questions while using Ernest is to determine when we need to retrain the model. We consider three different circumstances here: changes in code, changes in cluster behavior and changes in data.

Code changes: If different jobs use the same dataset, the cluster and dataset remain the same, but the computation being run changes. As Ernest treats the job being run as a black-box, we will need to retrain the model for any changes to the code. This can be detected by computing hashes of the binary files.

Variation in Machine Performance: One of the concerns with using cloud computing based solutions like EC2 is that there could be performance variations over time even when a job is using the same instance types and number of instances. We investigated if this was an issue by running two machine learning jobs GLM regression and NaiveBayes repeatedly on a cluster of 64 `r3.xlarge` instances. The time taken per run of each algorithm over a 24 hour period is shown in Figure 9. We see that the variation over time is very small for both workloads and the standard deviation is less than 2% of the mean. Thus we believe that Ernest models should remain relevant across relatively long time periods.

Changes in datasets: As Ernest uses small samples of the data for training, the model is directly applicable as the dataset grows. When dealing with newly collected data, there are some aspects of the dataset like the number of data items per block and the number of features per data item that should remain the same for the performance properties to be similar. As some of these properties might be hard to measure, our goal is to make the model building overhead small so that Ernest can be rerun for newly collected datasets.

5.2 Using Per-Task Timings

In the model described in the previous sections, we only measure the end-to-end running time of the whole job. Existing data processing frameworks already measure fine grained metrics [8, 3] and we considered integrating task-level metrics in Ernest. One major challenge we faced here is that in the BSP model a stage only completes when its last task completes. Thus rather than predicting the average task duration, we need to estimate the maximum task duration and this requires more complex non-parametric methods like Bootstrap [38]. Further, to handle cases where the number of tasks in a stage are greater than the number of cores available, we need adapt our estimate based on the number of waves [12] of tasks. We found that there were limited gains from incorporating task-level information given the additional complexity. While we continue to study ways to incorporate new features, we found that simple features used in predicting end-to-end completion time are more robust.

6 Evaluation

We evaluate how well Ernest works by using two metrics: the prediction accuracy and the overhead of training for long-running machine learning jobs. In experiments where we measure accuracy, or how close a prediction is to the actual job completion time, we use the ratio of the predicted job completion time to the actual job completion time $^{\text{Predicted Time}}/_{\text{Actual Time}}$ as our metric.

The main results from our evaluation are:

- Ernest's predictive model achieves less than 20% error on most of the workloads with less than 5% overhead for long running jobs.(§6.2)
- Using the predictions from Ernest we can get up to 4*x* improvement in price by choosing the optimal number of instances for the speech pipeline. (§6.3)

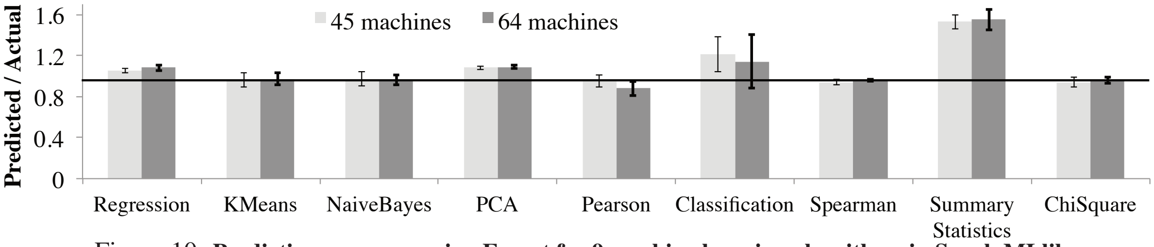
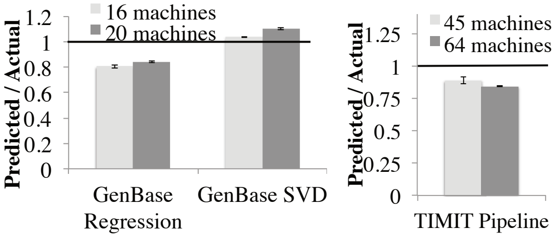


Figure 10: **Prediction accuracy using Ernest for 9 machine learning algorithms in Spark MLlib.**



(a) (b)

Figure 11: **Prediction accuracy for GenBase queries and TIMIT pipeline.**

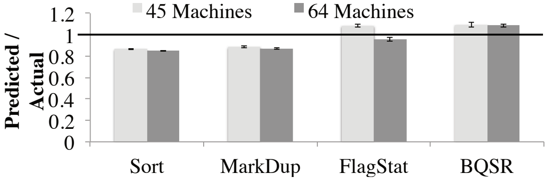


Figure 12: **Prediction accuracy for four transformations run using ADAM.**

- Given a training budget, experiment design improves accuracy by 30% – 50% for some workloads when compared to a cost-based approach. (§6.5)
- By extending the default model we are also able to accurately predict running times for sparse and randomized linear algebra operations. (§6.6)

6.1 Workloads and Experiment Setup

We use five workloads to evaluate Ernest. Our first workload consists of 9 machine learning algorithms that are part of MLlib [56]. For algorithms designed for dense inputs, the performance characteristics are independent of the data and we use synthetically generated data with 5 million examples. We use 10K features per data point for regression, classification, clustering and 1K features for the linear algebra and statistical benchmarks.

To evaluate Ernest on sparse data, we use `splice-site` and `kdda`, two of the largest sparse classification datasets that are part of LibSVM [28]. The `splice-site` dataset contains 10M data points with around 11M features and the `kdda` dataset contains around 6.7M data points with around 20M features. To see how well Ernest performs on end-to-end pipelines, we use GenBase, ADAM and a speech recognition pipeline (§2). We run regression and SVD queries from GenBase on the `Large` dataset [40] (30K genes for 40K patients). For ADAM we use the high coverage NA12878 full genome from the 1000 Genomes project [1] and run four transformations: sorting, marking duplicate reads, base quality score recalibration and quality validation. The speech recognition pipeline is run on the TIMIT [50] dataset using an implementation from KeystoneML [71]. All datasets other than the one for ADAM are cached in memory before the experiments begin and we do warmup runs to trigger the JVM's just-in-time compilation. We use `r3.xlarge` machines from Amazon EC2 (each with 4 vCPUs and 30.5GB memory) unless otherwise specified. Our experiments were run with Apache Spark 1.2. Finally all our predictions were compared against at least three actual runs and the values in our graphs show the average with error bars indicating the standard deviation.

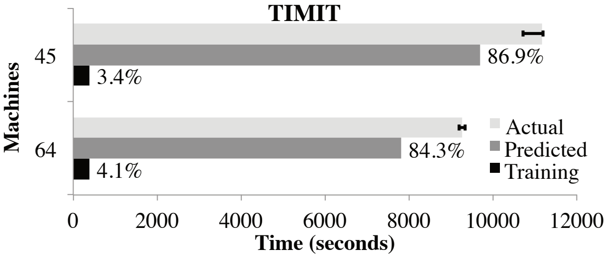


Figure 13: **Training times vs. accuracy for TIMIT pipeline running 50 iterations. Percentages with respect to actual running times are shown.**

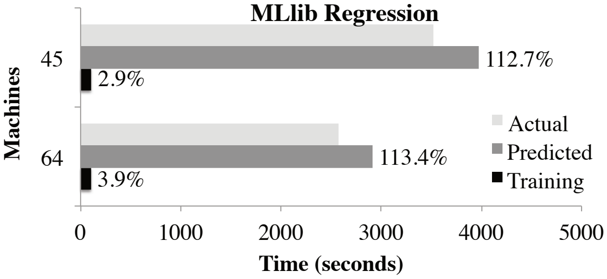


Figure 14: **Training times vs. accuracy for MLlib Regression running 500 iterations. Percentages with respect to actual running times are shown.**

6.2 Accuracy and Overheads

Prediction Accuracy: We first measure the prediction accuracy of Ernest using the nine algorithms from MLlib. In this experiment we configure Ernest to use between 1 and 16 machines for training and sample between 0.1% to 10% of the dataset. We then predict the performance for cases where the algorithms use the entire dataset on 45 and 64 machines. The prediction accuracies shown in Figure 10 indicate that Ernest's predictions are within 12% of the actual running time for most jobs. The two exceptions where the error is higher are the `summary statistics` and `glm-classification` job. In the case of `glm-classification`, we find that the training data and the actual runs have high variance (error bars in Figure 10 come from this) and that Ernest's prediction is within the variance of the collected data. In the case of summary statistics we have a short job where the absolute error is low: the actual running time is around 6 seconds while Ernest's prediction is around 8 seconds.

Next, we measure the prediction accuracy on GenBase and the TIMIT pipeline; the results are shown in Figure 11. Since the GenBase dataset is relatively small (less than 3GB in text files), we partition it into 40 splits, and restrict Ernest to use up to 6 nodes for training and predict the actual running times on 16 and 20 machines. As in the case of MLlib, we find the prediction errors to be below 20% for these workloads. Finally, the prediction accuracy for four transformations on ADAM show a similar trend and are shown in Figure 12. We note that the ADAM queries read input and write output to the *distributed filesystem* (HDFS) in these experiments and that these queries are also shuffle heavy. We find that Ernest is able to capture the I/O overheads and the reason for this is that the time to read / write a partition of data remains similar as we scale the computation.

Our goal in building Ernest is not to enforce strict SLOs but to enable low-overhead predictions that can be used to make coarse-grained decisions. We discuss how Ernest's prediction accuracy is sufficient for decisions like how many machines (§6.3) and what type of machines (§6.4) to use in the following sections.

Training Overheads: One of the main goals of Ernest is to provide performance prediction with low overhead. To measure the overhead in training we consider two long-running machine learning jobs: the TIMIT pipeline run for 50 iterations, and MLlib Regression with a mini-batch SGD solver run for 500 iterations. We configure Ernest to run 5% of the overall number of iterations during training and then linearly scale its prediction by the target number of iterations. Figures 13 and 14 show the times taken to train Ernest and the actual running times when run with 45 or 64 machines on the cluster. From the figures, we observe that for the regression problem the training time is below 4% of the actual running time and that Ernest's predictions are within 14%. For the TIMIT pipeline, the training overhead is less than 4.1% of the total running time. The low training overhead with these applications shows that Ernest efficiently handles long-running, iterative analytics jobs.

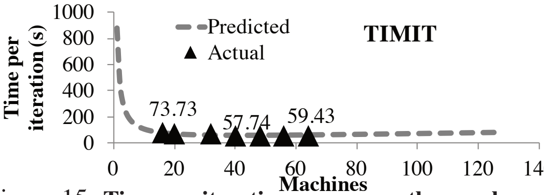


Figure 15: **Time per iteration as we vary the number of instances for the TIMIT pipeline. Time taken by actual runs are shown in the plot.**

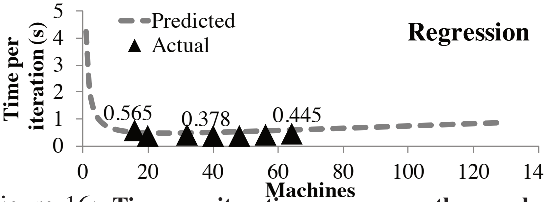


Figure 16: **Time per iteration as we vary the number of instances for MLlib Regression. Time taken by actual runs are shown in the plot.**

6.3 Choosing optimal number of instances

When users have a fixed-time or fixed-cost budget it is often tricky to figure out how many instances should be used for a job as the communication vs. computation trade-off is hard to determine for a given workload. In this section, we use Ernest's predictions to determine the optimum number of instances. We consider two workloads from the previous section: the TIMIT pipeline and GLM regression, but here we use subsets of the full data to focus on how the job completion time varies as we increase the number of machines to 64[1]. Using the same models trained in the previous section, we predict the time taken per iteration across a wide range of number of machines (Figures 15 and 16). We also show the actual running time to validate the predictions.

Consider a case where a user has a fixed-time budget of 1 hour (3600s) to say run 40 iterations of the TIMIT pipeline and an EC2 instance limit of 64 machines. Using Figure 15 and taking our error margin into account, Ernest is able to infer that launching 16 instances is sufficient to meet the deadline. Given that the cost of an `r3.xlarge` instance is $0.35/hour, a greedy strategy of using all the 64 machines would cost $22.4, while using the 16 machines as predicted by Ernest would only cost $5.6, a 4x difference. We also found that the 15% prediction error doesn't impact the decision as actual runs show that 15 machines is the optimum. Similarly, if the user has a budget of $15 then we can infer that using 40 machines would be faster than using 64 machines.

6.4 Choosing across instance types

We also apply Ernest to choose the optimal instance type for a particular workload; similar to the scenario above,

[1] We see similar scaling properties in the entire data, but we use a smaller dataset to highlight how Ernest can handle scenarios where the algorithm does not scale well.

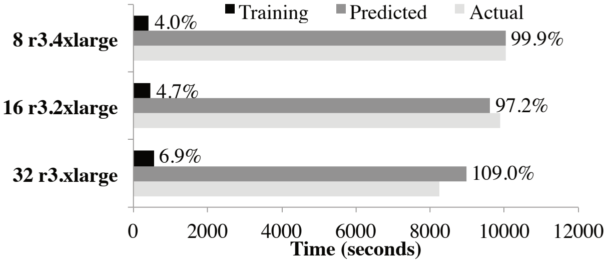
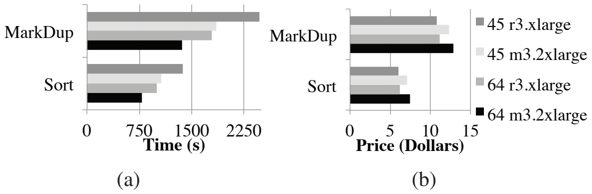


Figure 17: **Time taken for 50 iterations of the TIMIT workload across different instance types. Percentages with respect to actual running times are shown.**



(a)

(b)

Figure 18: **Time taken for Sort and MarkDup workloads on ADAM across different instance types.**

we can optimize for cost given a deadline or optimize for performance given a budget. As an example of the benefits of choosing the right instance type, we re-run the TIMIT workload on three instance types (`r3.xlarge`, `r3.2xlarge` and `r3.4xlarge`) and we build a model for each instance type. With these three models, Ernest predicts the expected performance on *same-cost* configurations, and then picks the cheapest one. Our results (Figure 17) show that choosing the smaller `r3.xlarge` instances would actually be 1.2*x* faster than using the `r3.4xlarge` instances, while *incurring the same cost*. Similar to the previous section, the prediction error does not affect our decision here and Ernest's predictions choose the appropriate instance type.

We next look at how choosing the right instance type affects the performance of ADAM workloads that read and write data from disk. We compare `m3.2xlarge` instances that have two SSDs but cost \$0.532 per hour and `r3.xlarge` instances that have one SSD and cost \$0.35 an hour[2]. Results from using Ernest on 45 and 64 machines with these instance types is shown in Figure 18. From the we can see that using `m3.2xlarge` instances leads to better performance and that similar to the memory bandwidth analysis (§2.2) there are non-linear price-performance trade-offs. For example, we see that for the mark duplicates query, using 64 `m3.2xlarge` instances provides a 45% performance improvement over 45 `r3.xlarge` instances while only costing 20% more.

6.5 Experiment Design vs. Cost-based

We next evaluate the benefits of using optimal experiment design in Ernest. We compare experiment design to a greedy scheme where all the candidate training data

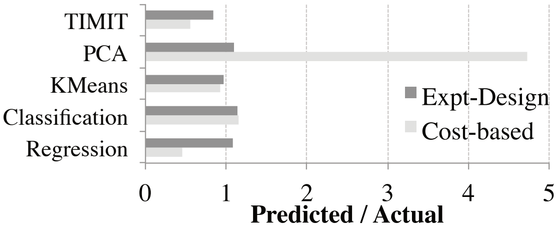


Figure 19: **Prediction accuracy when using Ernest vs. a cost-based approach for MLlib and TIMIT workloads.**

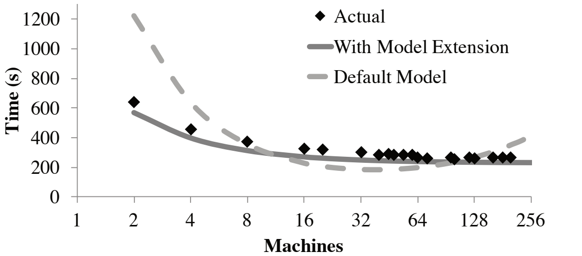


Figure 20: **Comparing KDDA models with and without extensions for different number of machines.**

points are sorted in increasing order of cost and we pick training points to match the cost of the points chosen in experiment design. We then train models using both configurations. A comparison of the prediction accuracy on MLlib and TIMIT workloads is shown in Figure 19.

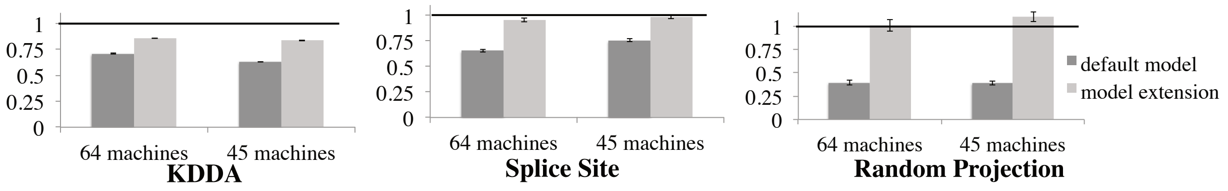
From the figure, we note that for some workloads (e.g. KMeans) experiment design and the cost-based approach achieve similar prediction errors. However, for the Regression and TIMIT workloads, Ernest's experiment design models perform 30% – 50% better than the cost-based approach. The cost-based approach fails because when using just the cheapest training points, the training process is unable to observe how different stages of the job behave as scale and number of machines change. For example, in the case of TIMIT pipeline, the cost-based approach explores points along a weak scaling curve where *both* data size and number of machines increase, thus it is unable to model how the Solver stage scales when the amount of data is kept constant. Ernest's optimal experiment design mechanism successfully avoids this and chooses the most useful training points.

6.6 Model Extensions

We also measure the effectiveness of the model extensions proposed in §3.4 on two workloads: GLM classification run on sparse datasets (§4.2) and a randomized linear algebra workload that has non-linear computation time [43]. Figure 21 shows the prediction error for the default model and the error after the model is extended: with a $\sqrt{n}$ term for the Sparse GLM and a $\frac{nlog^2 n}{mc}$ term which is the computation cost of the random projection. As we can see from the figure, using the appropriate model makes a significant difference in prediction error.

To get a better understanding of how different models can affect prediction error we use the KDDA dataset

[2] Prices as of September 2015



(a) (b) (c)

Figure 21: **Prediction accuracy improvements when using model extensions in Ernest. Workloads used include sparse GLM classification using KDDA, splice-site datasets and a random projection linear algebra job.**

and plot the predictions from both models as we scale from 2 to 200 machines (Figure 20). From the figure we can see that the extending the model with $\sqrt{n}$ ensures that the scaling behavior is captured accurately and that the default model can severely over-predict (at 2 machines and 200 machines) or under-predict (32 machines). Thus, while the default model in Ernest can capture a large number of workloads we can see that making simple model extensions can also help us accurately predict more complex workloads.

7 Related work

Performance Prediction: There have been a number of recent efforts at modeling job performance in datacenters to support SLOs or deadlines. Techniques proposed in Jockey [39] and ARIA [77] use historical traces and dynamically adjust resource allocations in order to meet deadlines. In Ernest we build a model with no historic information and try to minimize the amount of training data required. Bazaar [52] proposed techniques to model the network utilization of MapReduce jobs by using small subsets of data. In Ernest we capture computation and communication characteristics and use high level features that are framework independent. Projects like MRTuner [68] and Starfish [48] model MapReduce jobs at very fine granularity and set optimal values for options like memory buffer sizes etc. In Ernest we use few simple features and focus on collecting training data will help us maximize their utility. Finally scheduling frameworks like Quasar [36] try to estimate the scale out and scale up factor for jobs using the progress rate of the first few tasks. Ernest on the other hand runs the entire job on small datasets and is able to capture how different stages of a job interact in a long pipeline.

Query Optimization: Database query progress predictors [29, 57] solve a performance prediction problem similar to Ernest. Database systems typically use summary statistics [67] of the data like cardinality counts to guide this process. Further, these techniques are typically applied to a known set of relational operators. Similar ideas have also been applied to linear algebra operators [49]. In Ernest we use advanced analytics jobs where we know little about the data or the computation being run. Recent work has also looked at providing SLAs for OLTP [62] and OLAP workloads [46] in the cloud and some of our observations about variation across instance types in EC2 are also known to affect database queries.

Tuning, Benchmarking: Ideas related to experiment design, where we explore a space of possible inputs and choose the best inputs, have been used in other applications like server benchmarking [69]. Related techniques like Latin Hypercube Sampling have been used to efficiently explore file system design space [45]. Autotuning BLAS libraries [18] like ATLAS [31] also solve a similar problem of exploring a state space efficiently.

8 Future Work and Conclusion

In the future, we plan to study how statistical properties change in conjunction with the hardware. For example, in algorithms like HOGWILD! [66], the network latency between machines could affect the convergence rate. Further, based on our benchmarking experiments (§2) we see that there are a few key metrics which dictate the performance characteristics of a cluster. In the future we plan to study how we can integrate these metrics with the algorithm specific features used in Ernest.

In conclusion, the rapid adoption of advanced analytics workloads makes it important to consider how these applications can be deployed in a cost and resource-efficient fashion. In this paper, we studied the problem of performance prediction and show how simple models can capture computation and communication patterns. Using these models we have built Ernest, a performance prediction framework that intelligently chooses training points to provide accurate predictions with low overhead.

Acknowledgments

We would like to thank Ganesh Ananthanarayanan, Aurojit Panda, Kay Ousterhout, Evan Sparks, the NSDI reviewers and our shepherd Ranjita Bhagwan for their comments on earlier drafts of this paper. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, Guavus, HP, Huawei, Intel, Microsoft, Pivotal, Samsung, Schlumberger, Splunk, State Farm, Virdata and VMware.

References

[1] 1000 Genomes Project and AWS. http://aws.amazon.com/1000genomes/.

[2] Apache Mahout. http://mahout.apache.org/.

[3] Apache Spark: Monitoring and Instrumentation. http://spark.apache.org/docs/latest/monitoring.html.

[4] Apache spark, preparing for the next wave of reactive big data. http://goo.gl/FqEh94.

[5] Apache Spark: Submitting Applications. http://spark.apache.org/docs/latest/submitting-applications.html.

[6] Big data platform, hp haven. http://www8.hp.com/us/en/software-solutions/big-data-platform-haven/.

[7] Common crawl. http://commoncrawl.org.

[8] Hadoop History Server REST APIs. http://archive.cloudera.com/cdh4/cdh/4/hadoop/hadoop-yarn/hadoop-yarn-site/HistoryServerRest.html.

[9] Machine learning, microsoft azure. http://azure.microsoft.com/en-us/services/machine-learning/.

[10] MapReduce Tutorial. hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[11] V. S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D. A. Reed. An integrated compilation and performance analysis environment for data parallel programs. In *Supercomputing, 1995*.

[12] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *USENIX NSDI*, 2012.

[13] 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 *USENIX OSDI*, 2010.

[14] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390.

[15] Apache Hadoop NextGen MapReduce (YARN). Retrieved 9/24/2013, URL: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[16] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 213–223, Williamsburg, Virginia, USA.

[17] G. B. Berriman, G. Juve, E. Deelman, M. Regelson, and P. Plavchan. The application of cloud computing to astronomy: A study of cost and performance. In *Sixth IEEE International Conference on e-Science*, 2010.

[18] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Supercomputing 1997*, pages 340–347.

[19] I. Bird. Computing for the large hadron collider. *Annual Review of Nuclear and Particle Science*, 61:99–118, 2011.

[20] Blaise Barney. Message Passing Interface. https://computing.llnl.gov/tutorials/mpi/.

[21] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97.

[22] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, volume 20, pages 161–168. 2008.

[23] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

[24] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867, Prague, Czech Republic, June 2007.

[25] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing 2000*, Dallas, Texas, USA.

[26] J. P. Campbell Jr and D. A. Reynolds. Corpora for the evaluation of speaker recognition systems. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 829–832, 1999.

[27] B. L. Chamberlain, C. Lin, S.-E. Choi, L. Snyder, E. C. Lewis, and W. D. Weathersby. Zpl's wysiwyg performance model. In *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 50–61.

[28] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[29] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating progress of execution for SQL queries. In *SIGMOD 2004*, pages 803–814.

[30] M. Chowdhury and I. Stoica. Coflow: a networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36, 2012.

[31] R. Clint Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.

[32] A. Coates and A. Y. Ng. Learning feature representations with k-means. In *Neural Networks: Tricks of the Trade*, pages 561–580. Springer, 2012.

[33] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[34] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, pages 1232–1240, 2012.

[35] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1), 2008.

[36] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ASPLOS 2014*, pages 127–144.

[37] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[38] B. Efron. *The jackknife, the bootstrap and other resampling plans*, volume 38. SIAM, 1982.

[39] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Eurosys 2012*, pages 99–112.

[40] GenBase repository. https://github.com/mitdbg/genbase.

[41] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008. http://stanford.edu/~boyd/graph_dcp.html.

[42] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx, Mar. 2014.

[43] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[44] B. He, M. Yang, Z. Guo, R. Chen, W. Lin, B. Su, H. Wang, and L. Zhou. Wave computing in the cloud. In *HotOS*, 2009.

[45] J. He, D. Nguyen, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Reducing file system tail latencies with chopper. In *FAST 2015*, pages 119–133, Santa Clara, CA.

[46] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *VLDB 2011*, 4(11):1111–1122.

[47] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SOCC 2011*.

[48] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[49] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD 2015*.

[50] P.-S. Huang, H. Avron, T. N. Sainath, V. Sindhwani, and B. Ramabhadran. Kernel methods match deep neural networks on timit. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, page 6, 2014.

[51] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Eurosys 2007*.

[52] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *SOCC 2012*.

[53] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001–.

[54] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[55] X. Meng, J. Bradley, E. Sparks, and S. Venkataraman. ML Pipelines: A New High-Level API for MLlib. https://goo.gl/pluhq0, 2015.

[56] Apache Spark MLLib. https://spark.apache.org/mllib/.

[57] K. Morton, M. Balazinska, and D. Grossman. Paratimer: A progress indicator for mapreduce dags. In *SIGMOD 2010*, pages 507–518, Indianapolis, Indiana, USA.

[58] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP 2013*, pages 439–455.

[59] D. Narayanan. *Operating System Support for Mobile Interactive Applications*. PhD thesis, CMU, 2002.

[60] O. Niehorster, A. Krieger, J. Simon, and A. Brinkmann. Autonomic resource management with support vector machines. In *International Conference on Grid Computing (GRID '11)*, pages 157–164, 2011.

[61] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. J. Franklin, A. D. Joseph, and D. A. Patterson. Rethinking data-intensive science using scalable analytics systems. In *SIGMOD 2015*, pages 631–646.

[62] L. Ortiz, V. de Almeida, and M. Balazinska. Changing the Face of Database Cloud Services with Personalized Service Level Agreements. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, 2015.

[63] F. Pukelsheim. *Optimal design of experiments*, volume 50. SIAM, 1993.

[64] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2007.

[65] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC 1998*.

[66] B. Recht, C. Re, S. Wright, and F. Niu. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[67] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, 1979.

[68] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang. MRTuner: A Toolkit to Enable Holistic Optimization for MapReduce Jobs. *VLDB 2014*, 7(13).

[69] P. Shivam, V. Marupadi, J. Chase, T. Subramaniam, and S. Babu. Cutting corners: workbench automation for server benchmarking. In *USENIX ATC 2008*, pages 241–254.

[70] S. Sonnenburg and V. Franc. Coffin: A computational framework for linear svms. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 999–1006, 2010.

[71] E. Sparks. Announcing KeystoneML. https://amplab.cs.berkeley.edu/announcing-keystoneml, 2015.

[72] L. D. Stein et al. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[73] R. Taft, M. Vartak, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker. Genbase: A complex analytics genomics benchmark. In *SIGMOD 2014*, pages 177–188.

[74] J. Uszkoreit, J. M. Ponte, A. C. Popat, and M. Dubiner. Large scale parallel document mining for machine translation. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 1101–1109. Association for Computational Linguistics, 2010.

[75] R. A. Van De Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4):255–274, 1997.

[76] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *OSDI 2014*, pages 301–316.

[77] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *ICAC 2011*, pages 235–244, Karlsruhe, Germany.

[78] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Eurosys 2015*.

[79] W. Wang, L. Xu, and I. Gupta. Scale up Vs. Scale out in Cloud Storage and Graph Processing System. In *Proceedings of the 2nd IEEE Workshop on Cloud Analytics*, 2015.

[80] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *SOCC 2014*.

[81] W. Yoo, K. Larson, L. Baugh, S. Kim, and R. H. Campbell. Adp: Automated diagnosis of performance pathologies using hardware events. In *International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2012*, pages 283–294, London, England, UK.

[82] H.-F. Yu, H.-Y. Lo, H.-P. Hsieh, J.-K. Lou, T. G. McKenzie, J.-W. Chou, P.-H. Chung, C.-H. Ho, C.-F. Chang, Y.-H. Wei, et al. Feature engineering and classifier ensemble for KDD Cup 2010. *KDD Cup 2010*.

[83] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI 2012*.

Cliffhanger: Scaling Performance Cliffs in Web Memory Caches

Asaf Cidon[1], Assaf Eisenman[1], Mohammad Alizadeh[2], and Sachin Katti[1]

[1]Stanford University

[2]MIT CSAIL

ABSTRACT

Web-scale applications are heavily reliant on memory cache systems such as Memcached to improve throughput and reduce user latency. Small performance improvements in these systems can result in large end-to-end gains. For example, a marginal increase in hit rate of 1% can reduce the application layer latency by over 35%. However, existing web cache resource allocation policies are workload oblivious and first-come-first-serve. By analyzing measurements from a widely used caching service, Memcachier, we demonstrate that existing cache allocation techniques leave significant room for improvement. We develop Cliffhanger, a lightweight iterative algorithm that runs on memory cache servers, which incrementally optimizes the resource allocations across and within applications based on dynamically changing workloads. It has been shown that cache allocation algorithms underperform when there are performance cliffs, in which minor changes in cache allocation cause large changes in the hit rate. We design a novel technique for dealing with performance cliffs incrementally and locally. We demonstrate that for the Memcachier applications, on average, Cliffhanger increases the overall hit rate 1.2%, reduces the total number of cache misses by 36.7% and achieves the same hit rate with 45% less memory capacity.

1. INTRODUCTION

Memory caches like Memcached [13], Redis [4] and Tao [8] have become a vital component of cloud infrastructure. Major web service providers such as Facebook, Twitter, Pinterest and Airbnb have large deployments of Memcached, while smaller providers utilize caching-as-a-service solutions like Amazon ElastiCache [1] and Memcachier [3]. These applications rely heavily on caching to reduce web request latency, reduce load on backend databases and lower operating costs.

Even modest improvements to the cache hit rate impact performance in web applications, because reading from a disk or Flash-based database (like MySQL) is much slower than a memory cache. For instance, the hit rate of one of Facebook's Memcached pools is 98.2% [5]. Assuming the average read latency from the cache and MySQL is $200\mu s$ and $10ms$, respectively, increasing the hit rate by just 1% would reduce the read latency by over 35% (from $376\mu s$ at 98.2% hit rate to $278\mu s$ at 99.2% hit rate). The end-to-end speedup is even greater for user queries, which often wait on hundreds of reads [26].

Web caching systems are generally simple: they have a key-value API, and are essentially an in-memory hashtable spread across multiple servers. The servers do not have any centralized control and coordination. In particular, memory caches are *oblivious* to application request patterns and requirements. Memory allocation across slab classes[1] and across different applications sharing a cache server is based on fixed policies like first-come-first-serve or static reservations. The eviction policy, Least-Recently-Used (LRU), is also fixed.

By analyzing a week-long trace of a popular caching service, Memcachier, we show that existing first-come-first-serve cache allocation techniques can be greatly improved by applying workload aware resource policies. We show that for certain applications, the number of misses can be reduced by 99%.

We propose Cliffhanger, a lightweight iterative algorithm that runs on memory cache servers. Cliffhanger runs across multiple eviction queues. For each eviction queue, it determines the *gradient* of the hit rate curve at the current working point of the queue. It then incrementally allocates memory to the queues that would benefit the most from increased memory, and removes memory from the queues that would benefit the least.

Cliffhanger determines the hit rate curve gradient of each queue by leveraging shadow queues. Shadow queues are an extension of the eviction queue that only contain the keys of the requests, without the values. We show that the rate of hits in the shadow queue approxi-

[1] To avoid memory fragmentation, Memcached divides its memory into several *slabs*. Each slab stores items with size in a specific range (e.g., < 128B, 128-256B, etc.) [2]

mates the hit rate curve gradient. Cliffhanger differs from previous cache allocation schemes in that it does not require an estimation of the entire hit rate curves, which is expensive to estimate accurately. We also prove that although Cliffhanger is incremental and relies only on local knowledge of the hit rate curve, it performs as well as a system has knowledge of the entire hit rate curve.

Prior work has shown that existing cache allocation algorithms underperform when there are performance cliffs [6, 29, 35]. Performance cliffs occur when a small increase in memory creates an unexpectedly large change in hit rate. In other words, performance cliffs are regions in the hit rate curve where increasing the amount of memory for the queue accelerates the increase in hit rate. Memcachier's traces demonstrate that performance cliffs are common in web applications.

We propose a novel technique that deals with performance cliffs without having to estimate the entire hit rate curve. The technique utilizes a pair of small shadow queues, which allow Cliffhanger to locally search for where the performance cliff begins and ends. With this knowledge, Cliffhanger can overcome the performance cliff and increase the hit rate of the queue, by splitting the eviction queue into two and dividing the traffic across the two smaller queues [6]. We demonstrate that this algorithm removes performance cliffs in real time.

Cliffhanger runs on each memory cache server and iteratively allocates memory to different queues and removes performance cliffs in parallel. We show that the performance increase resulting from both algorithms is cumulative. Cliffhanger supports any eviction policy, including LRU, LFU or hybrid policies such as ARC [25].

We have built a prototype implementation of Cliffhanger in C for Memcached. We demonstrate that Cliffhanger can achieve the same hit rate as Memcached's default cache allocation using on average only 55% of the memory. Our micro-benchmark evaluation based on measurements at Facebook [5] shows that Cliffhanger incurs a negligible overhead in terms of latency and throughput. Since Cliffhanger uses fixed-sized shadow queues, its memory overhead is minimal: less than $500KB$ for each application. Applications typically use $50MB$ or more on each server.

2. BACKGROUND

Multi-tenant web caches need to allocate memory across multiple applications and across requests within applications. Typically individual applications are statically assigned a fixed amount of memory across multiple servers. Within each application, each request occupies a position in the queue based on its order. To avoid memory fragmentation, Memcached divides its memory into several *slabs*. Each slab stores items with size in a specific range (e.g., < 128B, 128-256B, etc.) [2]. Each slab has its own LRU queue. Once the memory cache is full, when a new request arrives, Memcached evicts the last item from its corresponding slab's LRU queue.

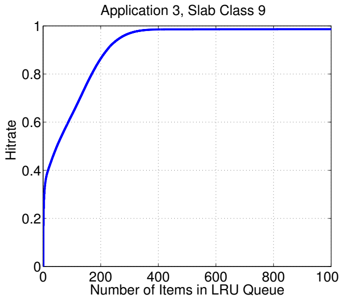


Figure 1: Hit rate curve for Application 3, Slab Class 9.

There are several problems with this first-come-first-serve approach. First, slab memory is divided greedily, without taking into account the slab sizes. Therefore, in many applications, the large requests will take up too much space at the expense of smaller requests. Second, when applications change their request distribution, certain slab classes may not have enough memory and their queues will be too short. Some Memcached implementations have tried to solve the second problem by periodically evicting a page of a slab class, and reassigning it to the corresponding slab class of new incoming requests [26, 30]. However, even these improved slab class allocation may suffer from sub-optimal slab class allocation, and they still treat large and small items equally. Third, if one slab class suffers from a very high rate of compulsory misses (i.e., many items never get accessed more than once), the web cache operator may prefer to shift its resources to a different slab class that can achieve a higher hit rate with the same amount of memory.

These problem are not specific to slab-based memory allocation like Memcached. They also occur in systems like RAMCloud [28, 31] that assign memory contiguously in a log (i.e., in a global LRU queue). Regardless of the memory allocation approach, memory is assigned to requests greedily: when new requests reach the cache server they are allocated memory on a first-come-first-serve basis, without consideration for the request size or the requirements of the requesting application.

2.1 The Cache Allocation Problem

As a motivating example, we describe how to optimize resource allocation across slab classes for a single application in Memcached, where the goal is to maximize the overall hit rate. The same technique can also be applied to prioritize items of different sizes in a log-structured cache and to optimize memory across applications. The problem can be expressed as an optimization:

$$\begin{aligned} \underset{m}{\text{maximize}} \quad & \sum_{i=1}^{s} w_i f_i h_i(m_i) \\ \text{subject to} \quad & \sum_{i=1}^{s} m_i \le M \end{aligned} \tag{1}$$

Where s is the number of slab classes, f_i is the frequency of GETs for each slab class, $h_i(m_i)$ is the hit rate of each slab class as a function of the its available memory (m_i), and M is the amount of memory reserved by the application on the Memcached server. In case different queues have different priorities, we can assign weights (w_i) to the different queues. For simplicity's sake, throughout the paper we assume that the weights of all queues are always equal to 1.

To accurately solve this optimization problem, we need to compute $h_i(m_i)$, or the *hit rate curve* for each slab class. Stack distances [24] enable the computation of the hit rate curve beyond the allocated memory size. The stack distance of a requested item is its rank in the cache, counted from the top of the eviction queue. For example, if the requested item is at the top of the eviction queue, its stack distance is equal to 1. If an item has never been requested before, its stack distance would be infinite. Figure 1 depicts the stack distances for a particular slab class in Memcachier, where the X axis is the size of the LRU queue required to achieve a certain hit rate. In Dynacache [10] we demonstrated how to solve Equation 1 by estimating stack distances and using a simple convex solver. Equation 1 can be extended to maximize the hit rate across applications, or to assign different weights to different request sizes and applications.

Estimating stack distances for each application and running a convex solver is expensive and complex. Computing the stack distances directly is $O(N)$, where N is the number of requests. Instead, we estimated the stack distances using the bucket algorithm presented in Mimir [32]. This technique is $O(\frac{N}{B})$, where B is the number of buckets (we used 100 buckets). However, this technique is not accurate when estimating stack distance curves with tens of thousands of items or more. In addition, since application workloads change over time, the solver would need to run frequently, typically on a different node than the cache server. Furthermore, each server would need to keep track of the stack distances of multiple applications. This introduces significant complexity to the simple design of web caches like Memcached.

3. TRACE ANALYSIS

In this section, we analyze the performance of Memcached's default resource allocation. We show that there is great potential to improve the hit rate of Memcached by optimizing memory across different request sizes within each application. We then investigate and characterize performance cliffs.

App	Slab Class	% GETs	Original % Misses	Dynacache % Misses
4	0	9%	0%	7.4%
4	1	91%	100%	92.6%
6	0	1%	0.1%	2%
6	2	70%	92.6%	0%
6	5	29%	0.1%	0.2%

Table 1: Misses in two applications compared by slab class. Application 4's misses were reduced by 6.3% and Application 6's were reduced by 91.7%.

Our analysis is based on a week-long trace of the top 20 applications (sorted by number of requests) running for a week on a server in Memcachier, a multi-tenant Memcached service. In Memcachier, each application reserves a certain amount of memory in advance, which is uniformly allocated across multiple Memcached servers comprising the Memcachier cache.

3.1 Resource Allocation in Memcachier

Figure 2 shows the hit rates and the number of misses that are reduced in Memcachier if we replace the default policy with the cache allocation using the Dynacache solver presented in Equation 1. We ran the solver on the week-long stack distances of each slab class. The results show that some applications benefit from a optimizing their memory across slab classes, and some do not. For example, the hit rate of Application 18 and 19 with full memory allocation is lower with the solver than the default algorithm. On the other hand, for some applications like Application 6, 14, 16 and 17, the number of misses is reduced by over 65%.

The solver's allocation is not optimal. This is due to several reasons. First, it assumes that all the hit rate curves are concave. We will explore this assumption later in the paper. Note that in Figure 2, the applications marked with asterisks are those that are not concave. Second, it requires that there be enough stack distance data points to accurately estimate the hit rate curve. If the requests for a slab class are too sparse, it will not be able to estimate its hit rate curve. Third, we ran the solver based on the trace of the entire week. If during that period the hit rate curves fluctuated considerably, the solver will not provide the ideal amount of memory for each slab class at any point in time. Due to these reasons, as we will show later, a heuristic dynamic cache allocation scheme can beat the solver.

3.2 Variance in Request Sizes

Table 1 demonstrates that the default scheme may assign too much memory to large slab classes, as evident

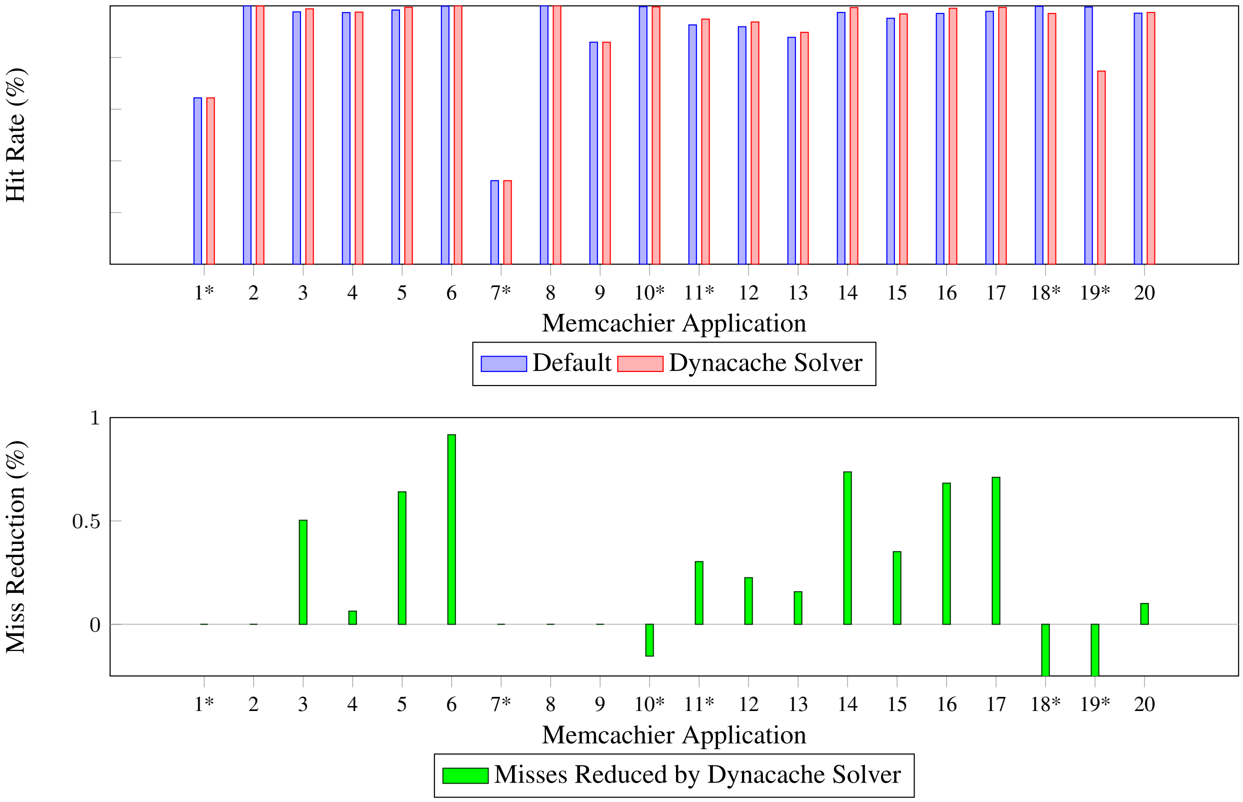


Figure 2: Hit rates and the number of misses reduced in Memcachier trace. Application 18 and 19's misses were increased by 13.6 and 51.5 times respectively. The applications that have an asterisk have performance cliffs.

Application	Original Hitrate	Log-structured Hitrate	Dynacache Solver Hitrate
3	97.7%	99.5%	98.8%
4	97.4%	97.8%	97.6%
5	98.4%	98.6%	99.4%

Table 2: Hit rates under log-structured memory and Dynacache solver.

in the case of applications 4 and 6. In application 6 this problem is much more severe, which is why the number of misses were reduced much more significantly by the Dynacache solver. The applications in the trace that did not see a significant hit rate improvement did not have much variance in terms of request size or were over-allocated memory.

The greedy nature of first-come-first-serve is not specific to Memcached's slab allocation scheme, it also occurs in contiguous Log Structured Memory (LSM). LSM [31] stores memory in a continuous log, rather than splitting it into slab classes, and it requires a log cleaner to run continuously to clear out stale items from the log and compact it. In memory caches, the benefit of LSM compared to slab classes would be the ability to run a global LRU queue, rather than having an LRU queue per slab class. To demonstrate this, we ran applications in three modes: with the default allocation of Memcached, the Dynacache solver allocation, and with a global LRU queue that simulates LSM. The global LRU simulates LSM with 100% memory utilization (such a scheme does not exist in practice). The results are displayed in Table 2. The results show that while LSM outperforms the Memcached slab allocator, in the case of application 5, even LSM running at 100% memory utilization may not perform as well as the optimized hit rate running on slab classes. The reason is that even in a global LRU queue large items may take space at the expense of small items.

3.3 Cross-application Performance

The Dynacache solver can be applied across applications running on the same memory cache server. To demonstrate the potential benefits of cross-application memory optimization, we applied the Dynacache solver to the top 5 applications in the trace. The results are displayed in Table 3. Note that in this example, we assigned each application the same weight. System operators can also assign different applications different weights in the optimization, for example, if certain applications belong

App	Original Memory Allocation %	Dynacache Solver Memory Allocation %	Original Hitrate	Dynacache Solver Hitrate
1	81%	69%	67.7%	67%
2	4%	13%	27.5%	38.6%
3	1%	1%	97.6%	97.6%
4	6%	8%	97.6%	98.1%
5	8%	9%	98.4%	98.5%

Table 3: Hit rates of the top 5 applications in the trace after we optimize their memory to maximize overall hit rate.

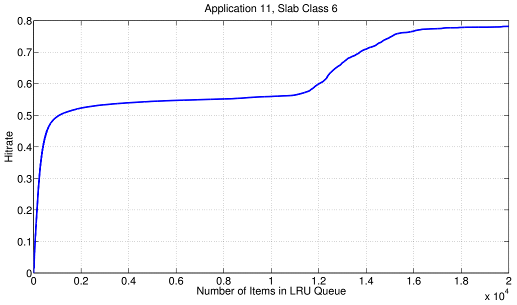


Figure 3: Examples of performance cliff in Memcachier traces.

to production systems and others do not.

3.4 Climbing Concave Hills

In order to solve Equation 1 we relied on estimating the entire hit rate curve. This requires a large number of stack distance data points and does not adapt to changing workloads. Instead, the same problem can be solved incrementally. If the gradient of each hit rate curve is known, we can incrementally add memory to the queue with the highest gradient (the one with the steepest hit rate curve), and remove memory from the queue with the lowest gradient. This process can be continued until shifting resources from one curve to the other will not result in overall hit rate improvement. This approach is called *hill climbing*.

The potential benefits of local hill climbing are that it can be conducted locally on each web-cache server, and that the algorithm is responsive to workload changes. If the hit rate curves change over time, the hill climbing algorithm incrementally responds. This leads us to ask: how can we measure the hit rate curve gradient locally without estimating the entire hit rate curve?

Our main insight is that the local hit rate curve can be measured using *shadow queues*. A shadow queue is an extension of an eviction queue that does not store the values of the items, only the keys. Items are evicted from the eviction queue into the shadow queue. For example, with an eviction queue of 1000 objects and a shadow queue of 1000 objects, when a request misses the eviction queue but hits the shadow queue, it means that if the eviction queue was allocated space for another 1000 objects, the request would have been a hit. The rate of hits in the shadow queue provides an approximation of the queue's local hit rate gradient.

3.5 Performance Cliffs

Hill climbing works well as long as the hit rate curves behave concavely and do not experience performance cliffs. This is true for some, but not for many web applications. Performance cliffs occur frequently in web applications: 6 out of the 20 top applications in our traces have performance cliffs. The applications in Figure 2 that have an asterisk are the ones with performance cliffs.

Figure 3 depicts the hit rate curve of a queue from the trace. Performance cliffs are thresholds where the hit rate suddenly changes as data fits in the cache. Cliffs occur, for example, with sequential accesses under LRU. Consider a web application that sequentially scans a 10 MB database. With less than 10 MB of cache, LRU will always evict items before they hit. However, with 10 MB of cache, the array suddenly fits and every access will be a hit. Therefore, increasing the cache size from 9 MB to 10 MB will increase the hit rate from 0% to 100%. Performance cliffs may hurt the hill climbing algorithm, because the algorithm will underestimate the gradient before a cliff, since it does not have knowledge of the entire hit rate curve. In the example of the queue in Figure 3, the algorithm can get stuck when the LRU queue has 10000 items.

Performance cliffs do not just hurt local-search based algorithms like hill climbing. They cause even bigger problems for solving optimization problems like the one described in Equation 1, since solvers assume that the hit rate curves do not have performance cliffs. For example, in application 19, due to the performance cliff, the solver approximates the hit rate curve to be lower than it is. This is why it fails to find the correct allocation for application 19, and significantly reduces its hit rate from 99.5% to 74.7%. In fact, optimal allocation is NP-complete without concave hit rate curves [6, 29, 35].

Resource allocation algorithms like Talus [6] and LookAhead [29] provide techniques to overcome performance cliffs for a single hit rate curve, but they require estimating the entire hit rate curve with stack distances. Since estimating stack distances introduces significant complexity and cost to web based storage systems, this leads us to ask: how can we overcome performance cliffs without estimating the entire hit rate curve?

4. DESIGN

In this section, we present the design of Cliffhanger, a hill-climbing resource allocation algorithm that runs on

Algorithm 1 Hill Climbing Algorithm

```
if request $\in$ shadowQueue(i) then
    queue(i).size = queue(i).size + credit
    chosenQueue = pickRandom({queues} - {queue(i)})
    chosenQueue.size = chosenQueue.size - credit
end if
```

each memory cache server and can scale performance cliffs. First, we describe the design of the hill climbing algorithm and show that it approximates the solution to the memory optimization problem. Second, we show how to overcome performance cliffs. Finally, we show how Cliffhanger climbs concave hit rate curves while navigating performance cliffs in parallel.

4.1 Hill Climbing Using Shadow Queues

The key mechanism we leverage to design our hill-climbing algorithm is shadow queues. Algorithm 1 depicts the hill-climbing algorithm, where $request$ is a new request coming into the cache server, $ShadowQueue(i)$ is the shadow queue of a particular queue (it can be the queue of a slab or a queue of an entire application), $pickRandom$ is a function that randomly picks one of the queues out the list of queues that we are optimizing. The algorithm is simple. Queue sizes are initialized so their capacity sums to the total available memory. When one of the shadow queues gets a hit, we increase its size by a constant credit, and randomly pick a different queue and decrease its size by the same constant credit. Once a queue reaches a certain amount of credits, it is allocated additional memory at the expense of another queue.

The intuition behind this algorithm is that the frequency of hits in the shadow queue is a function of the gradient of the local hit rate curve. In fact, we can prove that Algorithm 1 approximates the optimization described in Equation 1. Like before, we use the example of optimized memory across slab classes.

$$\begin{aligned} \underset{m}{\text{maximize}} \quad & \sum_{i=1}^{s} f_i h_i(m_i) \\ \text{subject to} \quad & \sum_{i=1}^{s} m_i \leq M \end{aligned}$$

As a reminder, s is the number of slab classes, f_i is the frequency of GETs, $h_i(m_i)$ is the hit rate as a function of the slab's memory, and M is the application's total amount of memory. Assume that $h_i(m_i)$ are increasing and concave. The Lagrangian for this problem is:

$$\mathcal{L}(m,\gamma) = \sum_{i=1}^{s} f_i h_i(m_i) - \gamma(\sum_{i=1}^{s} m_i - M)$$

The optimality condition is:

$$\begin{aligned} & f_i h_i'(m_i) = \gamma \text{ for } 1 \leq i \leq s \\ & \sum_{i=1}^{s} m_i = M \end{aligned} \tag{2}$$

Therefore, $f_i h_i'(m_i)$ is a constant (for any i), at the optimal solution. We show that with this algorithm, $f_i h_i'(m_i)$ is roughly constant in equilibrium for any i. To see why, consider the rate at which the credits for slab class i increase with a hit in its shadow queue:

$$f_i(h_i(m_i+\delta) - h_i(m_i)) \cdot \epsilon \approx f_i h_i'(m_i) \cdot \delta \cdot \epsilon$$

Here δ is the shadow queue size, and ϵ is the amount of credits we add to each queue when there is a hit in the shadow queue. The reason this approximation holds is that $f_i(h_i(m_i+\delta) - h_i(m_i))$ is the rate of hits that fall in the shadow queue. At the same time, the rate of credit decreases for class i is:

$$\frac{\sum_{j=1}^{s} f_j(h_j(m_j+\delta) - h_j(m_j)) \cdot \epsilon}{s} \approx$$
$$\frac{\sum_{j=1}^{s} f_j h_j'(m_j) \cdot \delta \cdot \epsilon}{s}$$

The reason this approximation holds is because when there is a hit in the shadow queue of any slab class j, we remove credits slab class i with probability $\frac{1}{s}$. In equilibrium the rate of credit increase and decrease have to be the same for every slab class. Therefore:

$$f_i h_i'(m_i) = \frac{\sum_{j=1}^{s} f_j h_j'(m_j) \cdot \delta \cdot \epsilon}{s} = \gamma$$

Since the right-hand side of the above equation does not depend on i, in equilibrium the gradients of each queue (normalized by the number of requests) are equal. This guarantees optimality (assuming concave hit rate curves) as shown in Equation 2.

4.2 Scaling Cliffs Using Shadow Queues

The hill climbing algorithm can get stuck in a sub-optimal allocation if the hit rates exhibit performance cliffs. We present for incrementally scaling performance cliffs. Our algorithm is inspired by Talus [6]. Talus introduced a queue partitioning scheme that scales performance cliffs, as long as the shape of the hit rate curve is known. Talus partitions a given queue to two smaller queues. By carefully choosing the ratio of requests it assigns to each queue and the size of the queues, Talus achieves the concave hull of the hit rate curve.

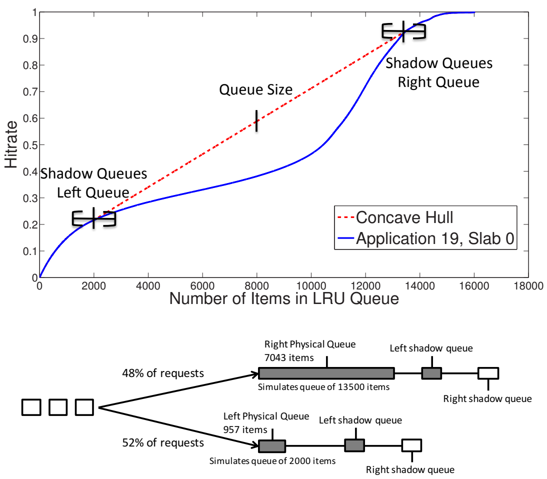


Figure 4: Visualization of shadow queues on Application 19, Slab Class 0.

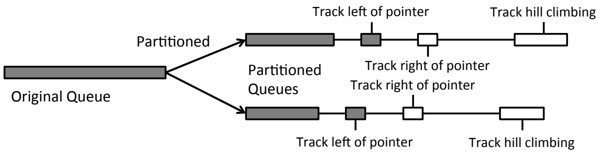


Figure 5: Illustration of Cliffhanger implementation. The darkly-colored sections represent the physical queue, which stores both the keys and the values of items, and the light sections are shadow queues.

We demonstrate how Talus works with the hit rate curve of the smallest slab class of Application 19. Figure 4 depicts the concave hull of the hit rate curve. Suppose the slab class is currently allocated 8000 items. Talus allows us to achieve a hit rate that is a linear interpolation between any two points in the hit rate curve, by simulating two queues of different sizes. For example, if we select the points that correspond to 2000 and 13500 items, Talus can trace the linear curve between these two points on the hit rate curve graph. It does so, by simulating a queue of size 2000 and a queue of size 13500. The key insight is that since each of the smaller queues get a fraction of the requests, it causes them to behave as if they were larger queues. In this example, if we split an 8000 item queue into a queue of 957 items and a queue of 7043 items, and split the requests between the two queues using a ratio of 0.48% and 0.52% respectively, the first queue will simulate a queue of 2000 items (957 items seeing a ratio of 0.48% of the requests), and the second queue will be of a simulated size of 13500 items (7043 items with a ratio of 52% of the requests). By partitioning the requests into these two queues, the application can achieve the hit rate of the concave hull. For more information, see Talus [6].

However, in our setting we do not know the entire hit rate curve. Therefore, to apply the Talus partitioning, we need to dynamically determine whether the current operating point is on a performance cliff of the hit rate curve, or in other words, whether it is in a convex section of the curve. We also need to find the two points in the curve that will provide anchors for the concave hull.

The key insight to determining whether a certain queue is in a convex section of the graph, is to approximate its second derivative. If the second derivative is positive, then the queue is currently in a convex area, a performance cliff. Similar to the hill-climbing algorithm, which locally approximates the first gradient with a shadow queue, to approximate the second derivative we use two shadow queues.

Each queue is split into two: a left and right physical queue. As long as the second derivative is negative (the function is concave), the left and right physical queues are the same size, and each receive half of the requests. Two evenly split queues behave exactly the same as one longer queue, since the frequency of the requests is halved for each queue, and the average stack distances of each request are also halved. Each one of the physical queues has its own shadow queue. The goal of these shadow queues is to find the points in the graph where the convex region ends. In the example of Application 19, these shadow queues are trying to locate points 2000 and 13500. In order to find the convexity region, each one of these shadow queues is in turn also split in half (right half and left half). The algorithm tracks whether each of the shadow queues receive hits in its right half or left half.

Algorithm 2 describes the cliff scaling algorithm. We initialize the algorithm by splitting the queue into half: a left and right physical queue, each with its own shadow queue, split in turn into two parts (the right and left half). The algorithm uses two pointers (right and left), which track the size of the simulated queues. The goal of the algorithm is to move the left and right pointers to the place in the hit rate curve where it starts and stops being convex. We initialize both of these pointers to the current size of the physical queue.

If the graph is in a convex point, the right pointer needs to be moved to the right, and the left pointer to the left, until it stops being convex. If each of the pointers are in a convex region of the graph, the rate of hits to the right of the pointer will be greater than to its left. When a request arrives to the server, we check if it hits the right or left shadow queue. If it hits the right half of the right shadow queue, we move the right pointer to the right. If it hits the left half, we move it to the left. Therefore, if the right pointer is in a convex region, the right pointer will move towards the top of the cliff. The left pointer moves in

Algorithm 2 Update Pointers

```
function Init
    ratio = 1/2
    rightPointer = leftPointer = queue.size
    updatePhysicalQueues(ratio)
end function
function updatePointers(request)
    if request ∈ rightShadowQueue then
        if request ∈ rightShadowQueue.rightHalf then
            rightPointer = rightPointer + credit
        end if
        if request ∈ rightShadowQueue.leftHalf AND
          rightShadowQueue.size > queue.size then
            rightPointer = rightPointer - credit
        end if
    end if
    if request ∈ leftShadowQueue then
        if request ∈ leftShadowQueue.rightHalf then
            leftPointer = leftPointer - credit
        end if
        if request ∈ leftShadowQueue.leftHalf AND
          leftShadowQueue.size < queue.size then
            leftPointer = leftPointer + credit
        end if
        ratio = computeRatio(queue.size,
          rightPointer, leftPointer)
        updatePhysicalQueues(ratio)
    end if
end function
```

Algorithm 3 Compute Ratios

```
function computeRatio(queue.size, rightPointer, leftPointer)
    distanceRight = rightPointer - queue.size
    distanceLeft = queue.size - leftPointer
    if distanceRight > 0 AND distanceLeft > 0 then
        requestRatio = distanceRight /
          (distanceRight + distanceLeft)
    else
        requestRatio = 0.5
    end if
end function
function updatePhysicalQueues(ratio)
    rightPhysicalQueue.size = rightPointer · (1 - ratio)
    leftPhysicalQueue.size = leftPointer · ratio
end function
```

the opposite direction. If it gets a hit to the right shadow queue, it moves left, and vice versa.

After adjusting the left and right pointers, Algorithm 3 updates the ratio of requests going to each physical queue, and their size. If the ratio is more than 0.5, more requests will be diverted to the left queue, and if the ratio is smaller than 0.5, more requests will be diverted to the right queue. As shown in Talus [6], the ratio is inverse to the distance of the position of the left and right shadow queues from the current operating point on the hit rate graph. In addition, the sizes of the right and left queues will also be updated based on this ratio. Their sum adds up to the current operating point (the queue size).

If Algorithm 2 does not see a performance cliff (i.e., a fully concave hit rate curve), the right and left pointers will not move from their starting points, because there will be more hits on the left halves of both queues than the right halves, since the hit rate curve is concave. Therefore, the physical queue will be split in half and each half will receive half of the requests, which will result in the same hit rate of the original queue.

Note that Algorithm 2 can only efficiently scale a single cliff. If there are multiple cliffs in the hit rate curve graph, the algorithm will either scale only one of them, or the right and left shadow queue positions will scale multiple cliffs. In any case, even if multiple cliffs are scaled, the resulting concave hull will be at a higher hit rate than at the original hit rate curve, since the cliffs are convex. In the Memcachier traces, we did not find an example of multiple performance cliffs in any of the hit rate curves.

4.3 Combining the Algorithms

So far, we've introduced two algorithms: first, the gradient approximation (hill climbing) algorithm allows us to iteratively optimize the resource allocation across multiple queues, and works well as long as they do not have performance cliffs. Second, the second derivative approximation (cliff scaling) algorithm allows us to "flatten" performance cliffs to their concave hulls.

Cliffhanger combines both algorithms by utilizing two shadow queues: a small shadow queue to approximate the second derivative and detect and mitigate performance cliffs, and a longer shadow queue appended to the shorter shadow queue to approximate the first gradient and perform hill-climbing. When we get a hit in the larger shadow queue, we assign credits to the entire slab class. When we get a hit at the end of the physical queue or in the small shadow queue, we adjust the two pointers and update the ratio and the split between the left and right queue. Cliffhanger runs on each memory cache server and does not require any coordination between different servers. In addition, it can support any eviction policy, including LRU, LFU and other hybrid schemes.

5. EVALUATION

In this section we present the implementation and evaluation of Cliffhanger on the Memcachier traces and a set of micro benchmarks.

5.1 Implementation

We implemented Cliffhanger on Memcached in C. The shadow queues were implemented on a separate hash and queue data structures. In order to measure the end-to-end performance improvement across the Memcachier applications, we re-ran the Memcachier traces and simulated the hit rate achieved by Cliffhanger. In order to stress the implementation, we ran our micro benchmarks on an Intel Xeon E5-2670 system with 32 GB of RAM and an SSD drive, using a micro bench mark workload gen-

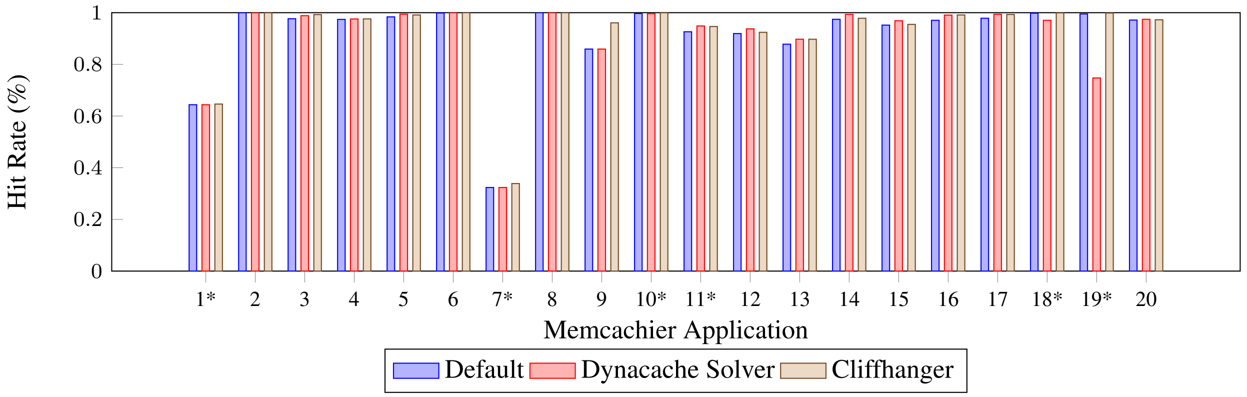


Figure 6: Hit rates of top 20 applications in Memcachier trace with Cliffhanger, compared to the Dynacache solver.

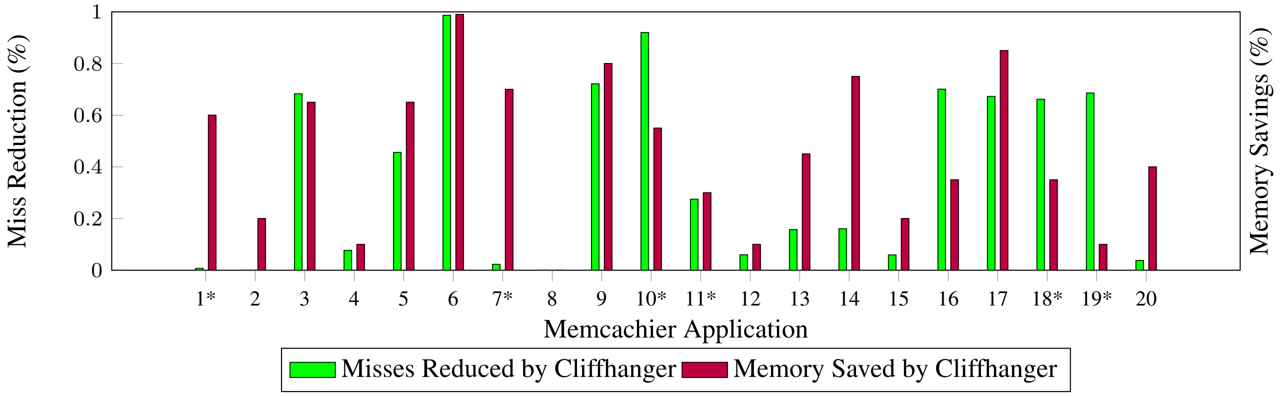


Figure 7: Miss reduction and amount of memory that can be saved to achieve the default hit rate of top 20 applications in Memcachier trace with Cliffhanger.

erated by Mutilate [19], a load generator that simulates traffic from the 2012 Facebook study [5].

Figure 5 is an illustration of the structure of the queues in Cliffhanger's implementation. Each queue is partitioned into two smaller queues (left and right queue). Each of the smaller queues needs to track whether items hit to the right or left of the pointers of the cliff scaling algorithm. In order to determine if an item hit to the left of the pointer, we do not need an extra shadow queue, since the section of the hit rate curve that is to the left of the pointer is already contained in the physical queue. To this end, our implementation tracks whether it sees hits in the last part of the queue (the last 128 items). In order to track hits to the right of the pointer, a 128 item shadow queue is appended after the physical queue. Finally, another shadow queue is appended to the end of each queue, to track hits for the hill climbing algorithm, since it requires a longer shadow queue.

The implementation only runs the cliff scaling algorithm when the queue is relatively large (over 1000 items), since it needs a large queue with a steep cliff to be accurate. It always runs the hill climbing algorithm, including on short queues. When it runs both of the algorithms in parallel, the 1 MB shadow queue used for hill climbing is partitioned into two shadow queues in proportion to the partition sizes (e.g., if the smaller partition is 0.4 MB and the larger one is 1.6 MB, the shadow queue will be split into 0.2 MB and 0.8 MB). To avoid thrashing, the cliff scaling algorithm resizes the two queues only when there is a miss (i.e., when a miss occurs we insert the new item into the queue that is larger).

5.2 Miss Reduction and Memory Savings

Figure 6 presents the hit rate of Cliffhanger, and Figure 7 presents the miss reduction of Cliffhanger compared to the default scheme, and the amount of memory that Cliffhanger requires to produce the same hit rate as the default scheme. Cliffhanger on average increases the hit rate by 1.2% and reduces the number of misses by 36.7%, and requires 55% of the memory to achieve the same hit rate as the default scheme.

For some of the applications, the reduction in misses is negligible (less than 10%). In these applications there is not much room for optimizing the memory alloca-

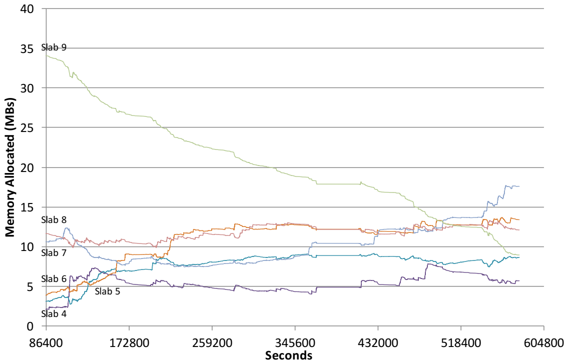


Figure 8: Memory allocated to slabs over time in Application 5, using Cliffhanger with shadow queues of 1 MB and 4 KB credits.

tion based on request sizes. For some applications, such as Application 5, 13 and 16, the hit rate with Cliffhanger is very similar to the hit rate of the Dynacache solver. In some applications, like applications 9, 18 and 19, Cliffhanger significantly outperforms the Dynacache solver. The reason that Cliffhanger outperforms the Dynacache solver in these applications is that it is an incremental algorithm, and therefore it can deal with hit rate curve changes even in queues that are relatively small. For the Dynacache solver to work well, it needs to profile a larger amount of data on the performance of the queue, otherwise it will not estimate the concave shapes of the curves accurately (for more information, see Dynacache [10]). In addition, application 19 has steep performance cliffs, which hurt the performance of the Dynacache solver.

5.3 Constants and Queue Sizes

Both the hill climbing algorithm and cliff scaling algorithms require the storage designer to determine the size of the shadow queues and the amount of credits we award each queue when it gets a hit to its shadow queue. For example, the behavior of the hill climbing algorithm is demonstrated in Figure 8. The figure depicts the memory allocation across slabs, when we use the hill climbing algorithm with shadow queues of 1 MB and 4 KB credits, and shows that it takes several days for the algorithm to shift memory across the different slabs. The reason it takes several days is that the request rate on each Memcachier server is relatively low: about 10,000 requests per second, across 490 different total applications.

We found little variance in the behavior of the hill climbing algorithm when we use shadow queues over 1 MB. The cliff scaling algorithm is more sensitive to the size of the shadow queues, since it measures the second gradient. We found that we achieve the highest hit rate when we use shadow queues of 128 items for the cliff scaling algorithm. We experimented with using different credit sizes for both algorithms, and found that 1-4 KB

Slab Class	Original Hitrate	Cliff Scaling Hitrate	Hill Climbing Hitrate	Combined Algorithm Hitrate
0	38.1%	44.8%	95.3%	98.3%
1	37.3%	45.6%	67.4%	69.1%
Total Hitrate	37.3%	45.5%	70.3%	72.1%

Table 4: Comparing the hit rates with the default scheme, with the hill climbing and cliff scaling algorithms.

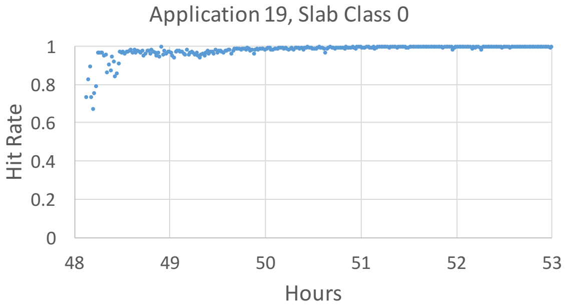


Figure 9: Hit rate of Application 19's Slab Class 0 under Cliffhanger, when the queue is in a region with a convex cliff.

provide the highest hit rates when we run the algorithms across an entire week. If we use significantly larger credits sizes, the algorithms may oscillate their memory allocation, which could cause thrashing across the queues.

5.4 Combined Algorithm Behavior

To better understand the affect of the hill climbing and cliff scaling algorithms, we focus on Application 19 that has steep performance cliffs in both slab classes. Table 4 depicts the results of running Cliffhanger on Application 19 when we use queues of 8000 items so that the hill-climbing algorithm gets stuck in the performance cliffs in both of its slab classes. We compare it to the default first-come-first-serve resource allocation, and to running only Algorithms 1 and 2 separately.

This demonstrates the algorithms have a cumulative hit rate benefit. The hill climbing algorithm's benefit is due to a long period where the application sends requests belonging to Slab Class 0, and then sends a burst of requests belonging to Slab Class 1. The reason the cliff scaling algorithm improves the hit rate, is that both slab class 0 and 1 are stuck in a performance cliff.

The behavior of the combined algorithms is demonstrated in Figure 9. The queue starts at a hit rate of about 70%. It takes about 30 minutes to stabilize until it reaches a hit rate of about 99.7%.

5.5 Comparison with LFU Schemes

Much of the previous work on improving cache hit rates focuses on allocating memory between LRU

Application	Original Hitrate	Facebook Hitrate	Cliffhanger + LRU Hitrate	Cliffhanger + Facebook Hitrate
3	97.7%	97.8%	99.3%	99.3%
4	97.4%	97.6%	97.6%	97.6%
5	98.4%	98.5%	99.1%	99.1%

Table 5: Hit rates with the Facebook eviction scheme.

Algorithm	Operation	Cache Hit	Cache Miss
Hill Climbing	GET	0%	1.4%
Hill Climbing	SET	0%	4.7%
Cliffhanger	GET	0.8%	1.4%
Cliffhanger	SET	0.8%	4.8%

Table 6: Average latency overhead when the cache is full.

and LFU queues. We compared the performance of Cliffhanger with two such schemes: the first is ARC [25], which splits each queue to an LRU and LFU queue and uses shadow queues to dynamically resize them based similar to our algorithm. The second scheme is used by Facebook, in which the first time a request hits it is inserted at the middle of the queue. When it hits a second time, it is inserted to the top of the queue.

We found that ARC did not provide any hit rate improvement in any of the applications of the Memcachier trace. We found that most recently used items that are ranked high in the LFU queue, are also ranked high by LRU. Therefore, the LFU queue never gets any hits in its shadow queue and does not get more memory.

The Facebook scheme performed better than LRU, but did not perform as well as Cliffhanger with LRU. Table 5 presents the results with Applications 3, 4 and 5.

5.6 Micro Benchmarks

We observed negligible latency and throughput overhead with high hit rates, such as the hit rate of most applications in Memcachier, since in the case of a hit, the shadow queue does not add any latency. To analyze the overhead of Cliffhanger under a worst-case scenario, we used synthetic trace where all keys are unique and all queries miss the cache. In this scenario the cache is always full, and every single request causes shadow queue allocations and evictions and all the GETs perform lookups in the shadow queue. We warm up the caches for 100 seconds to fill the eviction queues and shadow queues, and only then start measuring the latency and throughput. The experiment utilizes the same key and value distribution described by Facebook [5].

Table 6 shows that the average latency in this worst-case scenario was between 1.4%-4.8%, when the request missed. When the request hit there was no latency overhead with the hill climbing algorithm, since a hit does not require a lookup in the shadow queue, and 0.8% of latency with Cliffhanger, because we need to route the request between two physical queues. Table 7 presents the throughput overhead when the cache is full, which are identical when we are running the hill climbing algorithm alone and Cliffhanger. In any case, both Memcachier and Facebook are not CPU bound, but rather memory bound. Therefore, in both of these cases, increasing the average hit rate for applications at the expense of slightly decreased throughput at maximum CPU utilization is a favorable trade-off.

% GETs	% SETs	Throughput Slowdown
96.7%	3.3%	1.5%
50%	50%	3%
10%	90%	3.7%

Table 7: Throughput overhead when the cache is full and CPU bounded. The first row represents the GET/SET ratio in Facebook.

5.7 Memory Overhead

The memory overhead of Cliffhanger is minimal. The hill climbing algorithm uses shadow queues that represent 1 MB of requests. For example, with a 64 byte slab class the shadow queue will store 16384 keys, and with a $1KB$ slab class the shadow queue will store 1024 keys. The average key size is about 14 bytes. The cliff scaling algorithm uses a constant of 4 shadow queues (two left and right queues) of 128 items for each queue. Therefore, with the smallest slab class (64 bytes) the overhead will be $16384 + 128 \cdot 4 = 16896$ keys of 14 bytes for the smallest queue, which is about $200KB$ of extra memory for each queue. In Memcachier applications have 15 slab classes at most, and the overhead in the worst case will be $0.5MB$ of memory for each application.

6. RELATED WORK

There are two main bodies of related work. The first is previous work on improving the performance of web-based caches. The second is resource allocation techniques applied in other areas of caching and memory management, such as multi-core caches.

6.1 Web-based Memory Caches

Several systems improved the performance of memory cache servers by modifying their cache allocation and eviction policies. GD-Wheel [20] (GDW) uses the cost of recomputing the request in the database when prioritizing items in the cache eviction queue. This approach assumes the cache knows the recomputation cost in the database. Such information is not available to memory caches like Memcachier and Facebook, and would require changes to the memory cache clients. Regardless, Cliffhanger can be used with GDW as a replacement for LRU. Similar to Cliffhanger, GD-Size-Frequency (GDSF) [9] takes into account value size and frequency to replace LRU as a cache eviction algorithm for web proxy caches. GDSF relies on a global LRU queue, which is not available in Memcached, and on knowing the frequencies of each request. Unlike Cliffhanger, GDW and GDSF suffer from performance cliffs.

Mica [23], MemC3 [11] and work from Intel Labs [21] focus on improving the throughput of memory caches on multi-cores, by increasing concurrency and removing lock bottlenecks. While these systems offer significant throughput improvements over stock Memcached, they do not improve hit rates as a function of memory capacity. In the case of Facebook and Memcachier, Memcached is memory bound and not CPU bound.

Dynacache [10], Moirai [33, 34], Mimir [32] and Blaze [7] estimate stack distances and optimize resource allocation based on knowing the entire hit rate curve. Similar to Cliffhanger, Mimir approximates the hit rate curves using multiple buckets that contain only the keys and not the value sizes. Similarly, Wires et. al. profile hit rate curves using Counter Stacks [36] in order to better provision Flash based storage resources. All of these systems rely on estimation of the entire hit rate curve, which is generally more expensive and complex that local-search based methods like Cliffhanger, and do not deal with performance cliffs.

A recent study on the Facebook photo cache demonstrates that modifying LRU can improve web cache performance [14]. Twitter [30] and Facebook [26] have tried to improve Memcached slab class allocation to better adjust for varying item sizes, by periodically shifting pages from slabs with a high hitrate to those with a low hitrate. Unlike Cliffhanger, both of these schemes do not take into account the hit rate curve gradients and therefore would allocate too much memory to large requests.

Facebook [22], Zhang et al [37] and Fan et al [12] propose client-side proxies that provide better load-balacing and application isolation for Memcached clusters by choosing which servers to route requests to. While client-side proxies improve load-balancing, they do not control resource allocation within memory cache servers.

6.2 Cache Resource Allocation

Our work relies on results from multi-core cache partitioning. Talus [6] laid the groundwork for dealing with performance cliffs in memory caches, by providing a simple cache partitioning scheme that allows caches to trace the hit rate curve's concave hull, given knowledge of the shape of the convex portions of the hit rate graph. Talus relies on hardware utility monitors (UMONs) [29] to estimate stack distances and construct the hit rate curves. In contrast, Cliffhanger does not rely on profiling stack distance curves to trace the concave hull, and is therefore more lightweight and can incrementally adapt to changes in the hit rate curve profile of applications. LookAhead [29] is another algorithm that deals with performance cliffs. Instead of tracing the concave hull, it simply looks ahead in the hit rate curve graph, and allocates memory to applications after taking into account the affect of performance cliffs. Like Talus, it also relies on having knowledge of the entire hit rate curve.

There is an extensive body of working on workload aware eviction policies for multi-core systems that utilize shadow queues. A prominent example is ARC [25], which leverages shadow queues to dynamically allocate memory between LRU and LFU queues. Cliffhanger also leverages shadow queues in order to locate performance cliffs and dynamically shift memory across slab classes and applications. There are many other systems that try to improve on variants of LRU and LFU, including LRU-K [27], 2Q [16], LIRS [15] and LRFU [17, 18]. In addition, Facebook has implemented a hybrid scheme, where the first time a request is inserted into the eviction queue, it is not inserted at the top of the queue but in the middle. We have found that for the Memcachier traces, Facebook's hybrid scheme does provide hit rate improvements over LRU, and that ARC does not.

7. CONCLUSION

By analyzing a week-long trace from a multi-tenant Memcached cluster, we demonstrated that the standard hit rate of a data center memory cache can be improved significantly by using workload aware cache allocation. We presented Cliffhanger, a lightweight iterative algorithm, that locally optimizes memory allocation within and across applications. Cliffhanger uses a hill climbing approach, which allocates more memory to the queues with the highest hit rate curve gradient. In parallel, it utilizes a lightweight local algorithm to overcome performance cliffs, which have been shown to hurt cache allocation algorithms. We implemented Cliffhanger and evaluated its performance on the Memcachier traces and micro benchmarks. The algorithms introduced in this paper can be applied to other cache and storage systems that need to dynamically handle different request sizes and varying workloads without having to estimate global hit rate curves.

8. ACKNOWLEDGMENTS

We thank Amit Levy and David Terei, who helped us gather the traces from Memcachier. We also thank Nathan Bronson, Sathya Gunasekar, Anton Likhtarov, Ryan Stutsman, our shepherd, Mahesh Balakrishnan, and our reviewers for their valuable feedback.

References

[1] Amazon Elasticache. aws.amazon.com/elasticache/.

[2] Memcached. code.google.com/p/memcached/wiki/NewUserInternals.

[3] Memcachier. www.memcachier.com.

[4] Redis. redis.io.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[6] N. Beckmann and D. Sanchez. Talus: A simple way to remove cliffs in cache performance. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 64–75. IEEE, 2015.

[7] H. Bjornsson, G. Chockler, T. Saemundsson, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 59. ACM, 2013.

[8] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[9] L. Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[10] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association.

[11] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent Memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 385–398, 2013.

[12] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 23. ACM, 2011.

[13] B. Fitzpatrick. Distributed caching with Memcached. *Linux journal*, 2004(124):5, 2004.

[14] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 167–181. ACM, 2013.

[15] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, June 2002.

[16] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 439–450, 1994.

[17] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *ACM SIGMETRICS Performance Evaluation Review*, volume 27, pages 134–143. ACM, 1999.

[18] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, (12):1352–1361, 2001.

[19] J. Leverich. Mutilate. github.com/leverich/mutilate/.

[20] C. Li and A. L. Cox. GD-Wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, page 5. ACM, 2015.

[21] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 476–488. ACM, 2015.

[22] A. Likhtarov, R. Nishtala, R. McElroy, H. Fugal, A. Grynenko, and V. Venkataramani. Introducing McRouter: A memcached protocol router for scaling Memcached deployments, 2014. https://code.facebook.com/posts/296442737213493.

[23] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. *management*, 15(32):36, 2014.

[24] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[25] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[26] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[27] E. J. O'neil, P. E. O'neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.

[28] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[29] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006.

[30] M. Rajashekhar and Y. Yue. Twemcache. blog.twitter.com/2012/caching-with-twemcache.

[31] S. M. Rumble, A. Kejriwal, and J. K. Ousterhout. Log-structured memory for DRAM-based storage. In *FAST*, volume 1, page 16, 2014.

[32] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[33] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 174–181. ACM, 2015.

[34] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. Technical Report CSRG-626, Department of Computer Science, University of Toronto, 2015.

[35] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[36] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, A. Warfield, and C. Data. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 335–349. USENIX Association, 2014.

[37] W. Zhang, J. Hwang, T. Wood, K. Ramakrishnan, and H. Huang. Load balancing of heterogeneous workloads in Memcached clusters. In *9th International Workshop on Feedback Computing (Feedback Computing 14), Philadelphia, PA*, 2014.

FairRide: Near-Optimal, Fair Cache Sharing

Qifan Pu, Haoyuan Li (UC Berkeley), Matei Zaharia (MIT), Ali Ghodsi, Ion Stoica (UC Berkeley)

Abstract –

Memory caches continue to be a critical component to many systems. In recent years, there has been larger amounts of data into main memory, especially in shared environments such as the cloud. The nature of such environments requires resource allocations to provide both performance isolation for multiple users/applications and high utilization for the systems. We study the problem of fair allocation of memory cache for multiple users with shared files. We find that, surprisingly, no memory allocation policy can provide all three desirable properties (isolation-guarantee, strategy-proofness and Pareto-efficiency) that are typically achievable by other types of resources, e.g., CPU or network. We also show that there exist policies that achieve any two of the three properties. We find that the only way to achieve both isolation-guarantee and strategy-proofness is through *blocking*, which we efficiently adapt in a new policy called FairRide. We implement FairRide in a popular memory-centric storage system using an efficient form of *blocking*, named as *expected delaying*, and demonstrate that FairRide can lead to better cache efficiency (2.6× over isolated caches) and fairness in many scenarios.

1 Introduction

Caches are a crucial component of most computer systems, characterized by two features: their impact on application performance, and their limited size compared to the total amount of data. With in-memory caches increasingly being used for large-scale data processing clusters [4, 26] in addition to databases and key-value stores [19, 37, 10, 30, 8], caches also play a key role in today's multi-tenant cloud environments. In a shared environment with multiple users, however, the problem of managing caches becomes harder: how should a provider allocate space across multiple users, each of which wants to keep their own datasets in memory?

Unfortunately, traditional caching policies do not provide a satisfactory answer to this problem. Most cache management algorithms (e.g., LRU, LFU) have focused on *global efficiency* of the cache (Figure 1a): they aim to maximize the overall hit rate. Regardless of being commonly used in today's cache systems for cloud serving (Redis [9], Memcached [7]) and big data storage (HDFS Caching [5]), this has two problems in a shared environment. First, users who read data at long intervals may gain little or no benefit from the cache, simply because their data is likely to be evicted out of the memory. Second, applications can also easily *abuse* such systems by making spurious accesses to increase their access rate.

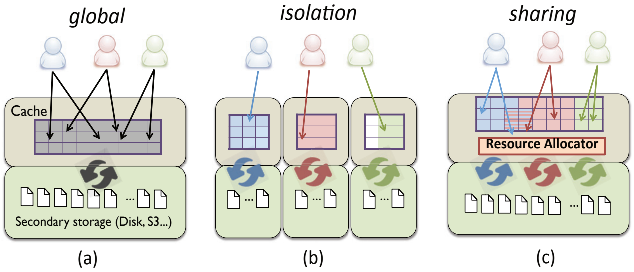


Figure 1: Different schemes. *Global*: single memory pool, agnostic of users or applications; *Isolation*: static allocations of memory among multiple users, possibly under-utilization (blank cells), no sharing; *Sharing*: allowing dynamic allocations of memory among users, and one copy of shared files (stripe cells).

There is no incentive to dissuade users from doing this in a cloud environment, and moreover, such shifts in the cache allocation can happen even with non-malicious applications. We show later that a strategic user can outperform a non-strategic user by 2.9×, simply by making spurious accesses to her files.

The other common approach is to have *isolated caches* for each user (Figure 1b). This gives each user performance guarantees and there are many examples in practice, e.g., hypervisors that set up separate buffer caches for each of its guest VMs, web hosting platforms that launch a separate `memcached` instance to each tenant. However, providing such performance guarantees comes at the cost of inefficient utilization of the cache.

This inefficiency is not only due to users not fully utilizing their allocated cache, but also because that a cached file can be accessed by multiple users at a time and isolating cache leads to multiple copies of such *shared files*. We find such *non-exclusive sharing* to be a defining aspect of cache allocation, while other resources are typically exclusively shared, e.g., a CPU time slice or a communication link can be only used by a single user at a time. In practice, there are a significant number of files shared across users in many workloads, e.g., we observe more than 30% files are shared by at least two users from a production HDFS log. Such sharing is likely to increase as more workloads move to multi-tenant environments.

In this paper, we study how to share cache space between multiple users that access shared files. To frame the problem, we begin by identifying desirable properties that we'd like an allocation policy to have. Building on common properties used in sharing of CPU and network resources [20], we identify three such properties:

	Isolation Guarantee	Strategy-Proofness	Pareto-Efficiency
global (e.g., LRU)	✗	✗	✓
max-min fairness	✓	✗	✓
FairRide	✓	✓	near-optimal
`None Exist`	✓	✓	✓

Table 1: **Summary of various memory allocation policies against three desired properties.**

- *Isolation Guarantee*: no user should receive less cache space than she would have had if the cache space were statically and equally divided between all users (i.e., assuming n users and equal shares, each one would get $1/n$ of the cache space). This also implies that the user's cache performance (e.g., cache miss ratio) should not be worse than isolation.

- *Strategy Proofness*: a user cannot improve her allocation or cache performance at the expense of other users by gaming the system, e.g., through spuriously accessing files.

- *Pareto Efficiency*: the system should be *efficient*, in that it is not possible to increase one user's cache allocation without lowering the allocation of some other user. This property captures operator's desire to achieve high utilization.

These properties are common features of allocation policies that apply to most resource sharing schemes, including CPU sharing via lottery or stride scheduling [39, 15, 35, 40], network link sharing via max-min fairness [28, 13, 17, 23, 33, 36], and even allocating multiple resources together for compute tasks [20]. Somewhat unexpectedly, there has been no equivalent policy for allocation of cache space that satisfies all three properties. As shown earlier, global sharing policies (Figure 1a) lack isolation-guarantee and strategy-proofness, while static isolation (Figure 1b) is not Pareto-efficient.

The first surprising result we find is that this deficiency is no accident: in fact, for sharing cache resources, *no policy can achieve all three properties*. Intuitively, this is because cached data can be shared across multiple users, allowing users to game the system by "free-riding" on files cached by others, or optimizing usage by caching popular files. This creates a strong trade-off between Pareto efficiency and strategy-proofness.

While no memory allocation policy can satisfy the three properties (Table 1), we show that there are policies that come close to achieving all three in practice. In particular, we propose FairRide, a policy that provides both isolation-guarantee (so it always performs no worse than isolated caches) and strategy-proofness (so users are not incentivized to cheat), and comes within 4% of global efficiency in practice. FairRide does this by aligning each user's benefit-cost ratio with her private preference, through *probabilistic blocking* (Section 3.4), i.e., probabilistically disallowing a user from accessing a cached file if the file is not cached on behalf of the user. Our proof in Section 5 shows that *blocking* is required to achieve strategy-proofness, and that FairRide achieves the property with minimal blocking possible.

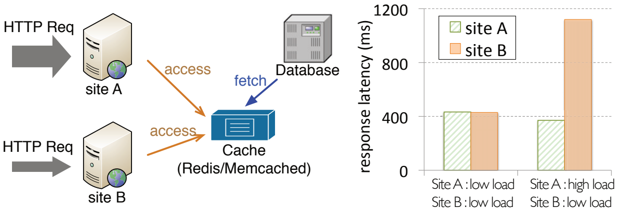
In practice, *probabilistic blocking* can be efficiently implemented using *expected delaying* (Section 4.1) in order to mitigate I/O overhead and to prevent even more sophisticated cheating models. We implemented FairRide on Tachyon [26], a memory-centric storage system, and evaluated the system using both cloud serving and big data workloads. Our evaluation shows that FairRide comes within 4% of global efficiency while preventing strategic users, meanwhile giving 2.6× more job runtime reduction over isolated caches. In a non-cooperative environment when users do cheat, FairRide outperforms max-min fairness by at least 27% in terms of efficiency. It is also worth noting that FairRide would support pluggable replacement policies as it still obeys each user's caching preferences, which allows users to choose different replacement policies (e.g., LRU, LFU) that best suit their workloads.

2 Background

Most of today's cache systems are oblivious to the entities (users) that access data: CPU caches do not care which thread accesses data, web caches do not care which client reads a web page, and in-memory based systems such as Spark [41] do not care which user reads a file. Instead, these systems aim to maximize system efficiency (e.g., maximize hit rate) and as a result favor users that contribute more to improve efficiency (e.g., users accessing data at a higher rate) at the expense of the other users.

To illustrate the unfairness of these cache systems, consider a typical setup of a hosted service, as shown in Figure 2a. We setup multiple hosted sites, all sharing a single Memcached [7] caching system to speed up the access to a back-end database. Assume the loads of A and B are initially the same. In this case, as expected, the mean request latencies for the two sites are roughly the same (see left bars in Figure 2b). Next, assume that the load of site A increases significantly. Despite the fact that B's load remains constant, the mean latency of its requests increases significantly (2.9×) and the latency for A's requests surprisingly drops! Thus, an increase in A's load improves the performance of A, but degrades the performance of B. This is because A accesses the data more frequently, and in response the cache system starts loading more results from A while evicting B's results.

While the example is based on synthetic web workload, this problem is very real, as demonstrated by the many questions posted on technical forums [6], on how



(a) (b)

Figure 2: (a) Typical cache setup for web servers. (b) Site B suffers high latency because unfair cache sharing.

to achieve resource isolation across multiple sites when using either Redis [9] or Memcached [7]. It turns out that none of the two popular caching systems provide any guarantee for performance isolation. This includes customized distributions from dominant cloud service providers, such as Amazon ElastiCache [1] and Microsoft Azure Redis Cache [3]. As we will show in Section 7, for such caching systems, it is easy for a strategic user to improve her performance and hurt others (with 2.9× performance gap) by making spurious access to files.

To provide performance isolation, the default answer in the context of cloud cache services today is to setup a separate caching instance per user or per application. This goes against consolidation and comes at a high cost. Moreover, cache isolation will eliminate the possibility of *sharing cached files*, which makes isolation even more expensive as there is a growing percentage of files to be shared. We studied a production HDFS log from a Internet company and observed 31.4% of files are shared by at least two users/applications. The shared files also tend to be more frequently accessed compared to non-shared files, e.g., looking at the 10% most accessed files, shared files account for as much as 53% of the accesses. The percentage of sharing can go even higher pair-wise: 22% of the users have at least 50% of their files accessed by another user. Assuming files are of equal sizes, we would need at least 31.4% more space if we assign isolated instances for each user, and even more cost on additional cache as the percentage of shared files in the working set is even larger.

Going back to the example in Figure 2, one possible strategy for *B* to reclaim some of its cache back would be to artificially increase its access rate. While this strategy may help *B* to improve its performance, it can lead to worse performance overall (e.g., lower aggregate hit rate). Worse yet, site *A* may decide to do the same: artificially increase its access rate. As we will show in this paper, this may lead to everyone losing out, i.e., everyone getting worse performance than when acting truthfully. Thus an allocation policy such as LRU is not strategy proof, as it does incentivize a site to misbehave to improve its performance. Furthermore, like with prisoner's dilemma, sites are incentivized to misbehave even if this leads to worse performance for everyone.

While in theory users might be incentivized to misbehave, a natural question is whether they are actually doing so in practice. The answer is "yes", with many real-world examples being reported in the literature. Previous works on cluster management [38] and job scheduling [20] have reported that users lie about their resource demands to game the system. Similarly, in peer-to-peer systems, "free-riding" is a well known and wide spread problem. In an effort to save bandwidth and storage, "free-riders" throttle their uplink bandwidth and remove files no longer needed, which leads to decreased overall performance [32]. We will show in Section 3 that shared files can easily lead to free-riding in cache allocation. Finally, as mentioned above, cheating in the case of caching is as easy as artificially increasing the access rate, for example, by running an infinite loop that accesses the data of interest, or just by making some random access. While some forms of cheating do incur certain cost or overhead (e.g., CPU cycles, access quota), the overhead is outweighed by the benefits obtained. On the one hand, a strategic user does not need many spurious accesses for effective cheating, as we will show in Section 7. If a caching system provides interfaces for users to specify file priorities or evict files in the system, cheating would be even simpler. On the other hand, many applications' performances are bottlenecked at I/O, and trading off some CPU cycles for better cache performance is worthwhile.

In summary, we argue that any caching allocation policy should provide the following three properties: (1) *isolation-guarantee* which subsumes performance isolation (i.e., a user will not be worse off than under static isolation), (2) *strategy-proofness* which ensures that a user cannot improve her performance and hurt others by lying or misbehaving, and (3) *Pareto efficiency* which ensures that resources are fully utilized.

3 Pareto Efficiency vs. Strategy Proofness

In this section we show that—under the assumption that the isolation-guarantee property holds—there is a strong trade-off between Pareto efficiency and strategy-proofness, that is, it is not possible to simultaneously achieve both in a caching system where files (pages) can be shared across users.

Model: To illustrate the above point, in the remainder of this section we consider a simple model where multiple users access a set of files. For generality we assume each user lets the cache system know the *priorities* in which her files can be evicted, either by explicitly specifying the priorities on the files or based on a given policy, such as LFU or LRU. For simplicity, in all examples, we assume that all files are of unit size.

```
    FUNC u.access(f)    // user u accessing file f
1:  if ($f \in Cache.fileSet$) then
2:      return CACHED_DATA;
3:  else
4:      return CACHE_MISS;
5:  end if

    FUNC cache(u, f)    // cache file f for user u
6:  while ($Cache.availableSize < f.size$) do
7:      $u1 = users.getUserWithLargestAlloc()$;
8:      $f1 = u1.getFileToEvict()$;
9:      if ($u1 == u$ and
10:         $u.getPriority(f1) > u.getPriority(f)$) then
11:         return CACHE_ABORT;
12:     end if
13:     $Cache.fileSet.remove(f1)$;
14:     $Cache.availableSize$ += $f1.size$;
15:     $u.allocSize$ -= $f1.size$;
16: end while
17: $Cache.fileSet.add(f)$
18: $Cache.availableSize$ -= $f.size$;
19: $u.allocSize$ += $f.size$;
20: return CACHE_SUCCEED;
```

Algorithm 1: Pseudocode for accessing and cahing a file under max-min fairness.

Utility: We define each user's utility function as the *expected cache hit rate*. Given a cache allocation, it's easy to calculate a user's expected hit rate by just summing up her access frequencies of all the files cached in memory.

3.1 Max-min Fairness

One of the most popular solutions to achieve efficient resource utilization while still providing isolation is *max-min fairness*.

In a nutshell, max-min fairness aims to maximize the minimum allocation across all users. Max-min fairness can be easily implemented in our model by evicting from the user with the largest cache allocation, as shown in Algorithm 1. When a user accesses a file, f, the system checks whether there is enough space available to cache it. If not, it repeatedly evicts the files of the users who have the largest cache allocation to make enough room for f. Note that the user from which we evict a file can be the same as the user who is accessing file f, and it is possible for f to not be actually cached. The latter happens when f has a lower caching *priority* than any of the other user's files that are already cached. At line 10 from Algorithm 1, the *user.getPriority*() is called to obtain *priority*. Note caching *priority* depends on the eviction policy. In the case of LFU, *priority* represents file's access frequency, while in the case of LRU it can represent the inverse of the time interval since it has been accessed. Similar to access frequency, *priority* need not to be static, but rather reflects an eviction policy's instantaneous preference.

If all users have enough demand, max-min fairness ensures that each user will get an equal amount of cache, and max-min fairness reduces to static isolation. However, if one or more users do not use their entire share, the unused capacity is distributed across the other users.

3.2 Shared Files

So far we have implicitly assumed that each user accesses different files. However, in practice multiple users may share the same files. For example, different users or applications can share the same libraries, input files and intermediate datasets, or database views.

The ability to share the same allocation across multiple users is a key difference between caching and traditional environments, such as CPU and communication bandwidth, in which max-min fairness has been successfully applied so far. With CPU and communication bandwidth, only a single user can access the resource that was allocated to her: a CPU time slice can be only used by a single process at a time, and a communication link can be used to send a single packet of a single flow at a given time.

A natural solution to account for shared files is to "charge" each user with a *fraction* of the shared file's size. In particular, if a file is accessed by k users, and that file is cached, each user will be charged with $1/k$ of the size of that file. Let $f_{i,j}$ denote file j cached on behalf of user i, and let k_j denote the number of users that have requested the caching of file j. Then, the total cache size *allocated* to user i, $alloc_i$, is computed as

$$alloc_i = \sum_j \frac{\text{size}(f_{i,j})}{k_j}. \tag{1}$$

Consider a cache that can hold 6 files, and assume three users. User 1 accesses files $A, B, C, \ldots$ user 2 accesses files $A, B, D, \ldots$, and user 3 accesses files $F, G, \ldots$. Assuming that each file is of unit size, the following set of cached files represent a valid max-min allocation: A, B, C, D, F, and G, respectively. Note that since files A and B are shared by the first two users, each of these users is only charged with half of the file size. In particular, the cache allocation of user 1 is computed as $\text{size}(A)/2 + \text{size}(B)/2 + \text{size}(C) = 1/2 + 1/2 + 1 = 2$. The allocation of user 2 is computed in a similar manner, while allocation of user 3 is simply computed as $\text{size}(F) + \text{size}(G) = 2$. The important point to note here is that while each user has been allocated the same amount of cache as computed by Eq. 1, users 1 and 2 get three files cached (as they get the benefit of sharing two of them), while user 3 gets only two.

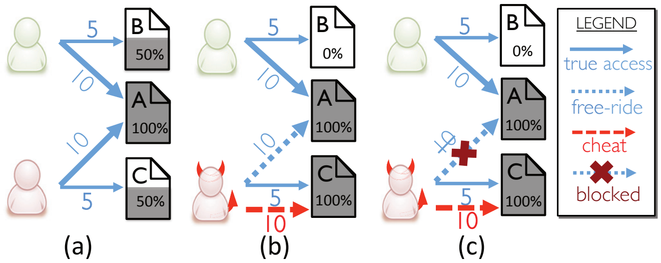


Figure 3: Example with 2 users, 3 files and total cache size of 2. Numbers represent access frequencies. (a). Allocation under *max-min fairness*; (b). Allocation under *max-min fairness* when second user makes spurious access (red line) to file C; (c). Blocking free-riding access (blue dotted line).

3.3 Cheating

While max-min fairness is strategy-proof when users access different files, this is no longer the case when files are shared. There are two types of cheating that could break strategy-proofness: (1) Intuitively, when files are shared, a user can "free ride" files that have been already cached by other users. (2) A thrifty user can choose to cache files that are shared by more users, as such files are more economic due to cost-sharing.

Free-riding To illustrate "free riding", consider two users: user 1 accesses files A and B, and user 2 accesses files A and C. Assume that size of cache is 2, and that we can cache a fraction of a file. Next, assume that every user uses LFU replacement policy and that both users access A much more frequently than the other files. As a result, the system will cache file A and "charge" each user by $1/2$. In addition, each user will get half of their other files in the cache, *i.e.,* half of file B for user 1, and file B for user 2, as shown in Figure 3(a). Each user gets a cache hit rate of $5 \times 0.5 + 10 = 12.5$[1] hits/sec.

Now assume user 2 cheats by spuriously accessing file C to artificially increase its access rate such that to exceed A's access rate (Figure 3(b)), effectively sets the *priority* of C higher than B. Since now C has the highest access rate for user 2, while A remains the most accessed file of user 1, the system will cache A for user 1 and C for user 2, respectively. The problem is that user 2 will still be able to benefit from accessing file A, which has already been cached by user 1. At the end, user 1 gets 10 hits/sec, and user 2 gets 15 hits/sec. In this way, user 2 free-rides on user 1's file A.

Thrifty-cheating To explain the kind of cheating where a user carefully calculates cost-benefits and then changes file priorities accordingly, we first define cost/(hit/sec) as the amount of budget cost a user pays to get 1 hit/sec access rate for a unit file. To optimize over the utility, which is defined as the total hit rate, a user's optimal strategy is not to cache the files that one has highest access frequencies, but the ones with lowest cost/(hit/sec). Compare a file of 100MB, shared by 2 users and another file of 100MB, shared by 5 users. Even though a user access the former 10 times/sec and the latter only 8 times/sec, it is overall economic to cache the second file (comparing 5MB/(hit/sec) vs. 2.5MB/(hit/sec)).

The consequence of "thrift-cheating", however, is more complicated. As it might appear to improve user and system performance at first glance, it doesn't lead to an equilibrium where all users are content about their allocations. This can cause users to constantly game the system which leads to a worse outcome.

In the above examples we have shown that due to another user cheating, one can experience utility loss. A natural question to ask is, how bad could it be? i.e. What is the upper bound a user can lose when being cheated? By construction, one can show that for two-user cases, a user can lose up to 50% of cache/hit rate when all her files are shared and "free ridden" by the other strategic user. As the free-rider evades charges of shared files, the honest user double pays. This can be extended to a more general case with n ($n > 2$) users, where loss can increase linearly with the number of cheating users. Suppose that cached files are shared by n users, each user pays $\frac{1}{n}$ of the file sizes. If $n-1$ strategic users decide to cache other files, the only honest user left has to pay the total cost. In turn, the honest user has to evict at most $(\frac{n-1}{n})$ of her files to maintain the same budget.

It is also worth mentioning that for many applications, moderate or even minor cache loss can result in drastic performance drop. For example, in many file systems with overall high cache hit ratio, the effective I/O latency with caching could be approximated as $T_{IO} = Ratio_{miss} Latency_{miss}$. A slight difference in the cache hit ratio, e.g. from 99.7% to 99.4%, means 2× I/O average latency drop! This indeed necessitates strategy-proofness in cache policies.

3.4 Blocking Access to Avoid Cheating

At the heart of providing strategy-proofness is this question of how free-riding can be prevented. In the previous example, user 2 was incentivized to cheat because she was able to access the cached shared files regardless her access patterns. Intuitively, if user 2 is blocked from accessing files that she tries to free-ride, she will be dis-incentivized to cheat.

Applying blocking to our previous example, user 2 will not be allowed to access A, despite the fact that user 1 has already cached A (Figure 3(c)). The system blocks

[1] When half of a file is in cache, half of the page-level accesses to the file will result in cache miss. Numerically, it is the equal to missing the entire file 50% of the time. So hit rate is calculated as access rate multiplied by percentage cached.

user 2 but not user 1 because user 1 is the sole person who pays the cache. As a result, user 2 gets only 1 cache size with a less important file C.

As we will show in Section 5 this simple scheme is strategy-proof. On the other hand, this scheme is unfortunately not Pareto efficient by definition, as the performance (utility) of user 2 can be improved without hurting user 1 by simply letting user 2 access file A.

Furthermore, note that it is not necessary to have a user cheating to arrive at the allocation in Figure 3. Indeed, user 2 can legitimately access file C at a much higher rate than A, In this case, we get the same allocation—file A is cached on behalf of user 1 and file C is cached on behalf of user 2—with no user cheating. Blocking in this case will reduce the system utilization by punishing a well-behaved user.

Unfortunately, the cache system cannot differentiate between a cheating and a well-behaved user, so it is not possible to avoid the decrease in the utilization and thus the violation of Pareto efficiency, even when every user in the system is well-behaved.

Thus, in the presence of shared files, with max-min fairness allocation we can achieve either Pareto efficiency or strategy-proofness, but not both. In addition, we can trade between strategy-proofness and Pareto efficiency by blocking a user from accessing a shared file if that file is not in the user's cached set of files, even though that file might have been cached by other users.

In Section 5, we will show that this trade-off is more general. In particular, we show that in the presence of file sharing there is no caching allocation policy that can achieve more than two out of the three desirable properties: isolation-guarantee, strategy-proofness, and Pareto efficiency.

4 FairRide

In this section, we describe FairRide, a caching policy that extends max-min fairness with *probabilistic blocking*. Different from max-min fairness, FairRide provides isolation-guarantee and strategy-proofness at the expense of Pareto-efficiency. We use *expected delaying* to implement the conceptual model of *probabilistic blocking*, due to several system considerations.

Figure 4 shows the control logic for a user i accessing file j under FairRide. We will compare it with the pseudo-code of max-min fairness, Algorithm 1. In max-min, a user i can directly access a cached file j, as long as j is cached in memory. While with FairRide, there is an chance that the user might get blocked for accessing the cached copy. This is key to making FairRide strategy-proof and the *only* difference with max-min fairness, which we prove in Section 5. The chance of blocking is not an arbitrary probability, but is set at $\frac{1}{n_j+1}$, where n_j is the number of other users caching the file. We will prove in Section 5 that this is the only and minimal blocking probability setting that will make a FairRide strategy-proof.

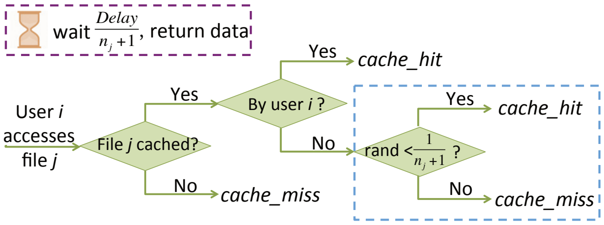

wait $\frac{Delay}{n_j+1}$, return data
User i accesses file j
File j cached?
Yes
No
cache_miss
By user i ?
Yes
cache_hit
No
rand < $\frac{1}{n_j+1}$?
Yes
cache_hit
No
cache_miss

Figure 4: With FairRide, a user might be blocked to access a cached copy of file if the user does not pay the storage cost. The blue box shows how this can be achieved with *probabilistic blocking*. In system implementation, we replace the blue box with the purple box, where we instead delay the data response.

Consider again the example in Figure 3. If user 2 cheats and makes spurious access to file C, file A will be cached on behalf of user 1. In that case, FairRide recognizes user 2 as a non-owner of the file, and user 2 has $\frac{1}{2}$ chance to access directly from the cache. So user 2's total expected hit rate becomes $5+10\times\frac{1}{2}=10$, which is worse than 12.5 before without cheating. In this way, FairRide discourages cheating and makes the policy strategy-proof.

4.1 Expected Delaying

In real systems, *probabilistic blocking* could not thoroughly solve the problem of cheating, as now a strategic user can make even more accesses in hope that one of the accesses is not blocked. For example, if a user is blocked with a probability of $\frac{1}{2}$, he can make three accesses so to reduce the likelihood of being blocked to $\frac{1}{8}$. In addition, *blocking* itself is not an ideal way to implement in a system as it further incurs unnecessary I/O operations (disk, network) for blocked users. To address this problem, we introduce *expected delaying* to approximate the expected effect of *probabilistic blocking*. When a user tries to access an in-memory file that is cached by other users, the system delays the data response with certain wait duration. The wait time should be set as the expected delay a user would experience if she's probabilistically blocked by the system. In this way, it is impossible to get around the delaying effect, and the system does not have to issue additional I/O operations. The theoretically equivalent wait time could be calculated as $t_{wait} = Delay_{mem} \times (1-p_{block}) + Delay_{disk,network} \times p_{block}$, where p_{block} is the blocking probability as described above, and $Delay_x$ being the access latency of medium x. As memory access latency is already incurred during data read time, we simply set the wait time to be $Delay_{disk,newtwork} \times p_{block}$. We will detail how we measure the secondary storage delay in Section 6.

5 Analysis

In this section, we prove that the general trade-off between the three properties is fundamental with existence of file sharing. Next in Section 5.2, we also show by proof that FairRide indeed achieves strategy-proof and isolation-guarantee, and that FairRide uses most efficient blocking probability to achieve strategy-proofness.

5.1 The SIP theorem

We state the following **SIP theorem**: *With file sharing, no cache allocation policy can satisfy all three following properties: strategy-proofness (S), isolation-guarantee (I) and Pareto-efficiency (P).*

Proof of the SIP theorem
The three properties are defined as in Section 1, and we use total hit rate as the performance metric. Reusing the example setup in Figure 3(a), we now examine a *general* policy P. The only assumption of P is that P satisfies isolation-guarantee and Pareto-efficiency, and we shall prove that such policy P must not be strategy-proof, i.e. a user can cheat to improve under P. We start with the case when no user cheats for Figure 3(a). Let y_1, y_2 be user 1 and 2's total hit rate:

$$y_1 = 10x_A + 5x_B \tag{2}$$

$$y_2 = 10x_A + 5x_C \tag{3}$$

Where x_A, x_B, x_C are fractions of the each file A, B, C cached in memory.[2] Because $x_A + x_B + x_C = 2$, and $y_1 + y_2 = 15x_A + 5(x_A + x_B + 5x_c)$, it's impossible for $y_1 + y_2 > 25$, or, for both y_1 and y_2 to be greater than 12.5. As the two users have symmetric access patterns, we assume $y_2 < 12.5$ without loss of generality.

Now if user 2 cheats and increases her access rate of file C to 30, we can prove that she can get a total rate of 13.3, or $y_2 > 13.3$. This is partly because the system has to satisfy a new isolation guarantee:

$$y_2' = 10x_A + 30x_C > 30 \tag{4}$$

It must hold that $x_C > \frac{2}{3}$, because $x_A \leq 1$. Also, because $x_C \leq 1$ and $x_A + x_B + x_C = 2$, we have $x_A + x_B \geq 1$ to achieve Pareto-efficiency. For the same reason, $x_A = 1$ is also necessary as it's strictly better to cache file A over file B for both users. Plugging $x_A = 1, x_C > \frac{2}{3}$ back to user 2's actual hit rate calculation (Equation 3), we get $y_2 > 13.3$.

So far, we have found a cheating strategy for a user 2 to improve her cache performance and hurt the other user. This is done under a general policy P that assumes only isolation-guarantee and Pareto-efficiency but nothing else. Therefore, we can conclude that any policy P that satisfies the two properties cannot achieve strategy-proofness. In other words, no policy can achieve all three properties simultaneously. This ends the proof for the SIP theorem.

5.2 FairRide Properties

We now examine FairRide (as described in Section 4) against three properties.

Theorem FairRide *achieves isolation-guarantee.*
Proof Even if FairRide does complete blocking, in which each user gets strictly less memory cache, the amount of cache a user accesses is: $Cache_{total} = \sum_j size(file_j)$, j for all the files the user caches. Because FairRide splits the charges of shared files across all users, a user's allocation budget is spent up by: $Alloc = \sum_j \frac{size(file_j)}{n_j}$, with n_j being the number of users sharing $file_j$. Combining the two equations we can easily derive that $Cache_{total} > Alloc$. As $Alloc$ is also what a user can get in *isolation*, we can conclude that the amount of memory a user can access is always bigger than *isolation*. Likewise, we can prove the total hit rate user gets with FairRide is greater than *isolation*.

Theorem FairRide *is strategy-proof.*
Proof We will sketch the proof using cost-benefit analysis, following the line of reasoning in Section 3.3. With *probabilistic blocking*, a user i can access a file j without caching it with a probability of $\frac{n_j}{n_j+1}$. This means that the benefit resulted from caching is the $increased\_rate$, equal to $freq_{ij}\frac{1}{n_j+1}$. The cost is $\frac{1}{n_j+1}$ for the joining user, with n_j other users already caching it. Dividing the two, the benefit-cost ratio is equal to $freq_{ij}$, user i's access frequency of file j. As a user is incentivized to cache files based on the descending order of benefit-cost ratio, this results in caching files based on actual access frequencies, rather than cheating. In other words, FairRide is incentive-compatible and allows users to perform truth-telling.

Theorem FairRide*'s uses lower-bound blocking probabilities for achieving strategy-proofness.*
Proof Suppose a user has 2 files: f_j, f_k with access frequencies of $freq_j$ and $freq_k$. We use p_j and p_k to denote the corresponding blocking probabilities if the user chooses not to cache the files. Then the benefit-cost ratios for the two files are $freq_j p_j (n_j + 1)$ and $freq_k p_k (n_k + 1)$, n_j and n_k being the numbers of other users already caching the files. For the user to be truth-telling for whatever $freq_j$, $freq_k$, n_j or n_k, we must have $\frac{p_j}{p_k} = \frac{n_k+1}{n_j+1}$. Now p_j and p_k can still be arbitrarily small or big, but note $p_j(p_k)$ must be 1 when

[2] We use fractions only for simplifying the proof. The theorem holds when we can only cache a file/block in its entirety.

$n_j(n_k)$ is 0, as no user is caching file $f_j(f_k)$. Putting $p_j = 1, n_j = 0$ into the equation we will have $p_k = \frac{1}{n_k+1}$. Similarly, $p_j = \frac{1}{n_j+1}$. Thus we show that FairRide's blocking probabilities are the only probabilities that can provide strategy-proofness in the general case (assuming any access frequencies and sharing situations). The only probabilities are also the lower-bound probabilities.

6 Implementation

FairRide is evaluated through both system implementation and large-scale trace simulations. We have implemented FairRide allocation policy on top of Tachyon [26], a memory-centric distributed storage system. Tachyon can be used as a caching system and it supports in-memory data sharing across different cluster computation frameworks or applications, e.g. multiple Hadoop Mapreduce [2] or Spark [41] applications.

Users and Shares Each application running on top of Tachyon with FairRide allocation has a FairRide client ID. Shares for each user can be configured. When shares are changed during system uptime, cache allocation is re-allocated over time, piece by piece, by evicting files from the user who uses most atop of her share, i.e., $argmax_i(Alloc_i - Capacity * Share_i)$, thus converging to the configured shares eventually.

Pluggable Policy Because FairRide obeys each user's individual caching preferences, it can apply a two-level cache replacement mechanism. It first picks the user who occupies the most cache in the system, and then finds the least preferred file from that user to evict. This naturally enables "pluggable policy", allowing each user to pick a replacement policy best fit for her workload. Note this would not be possible for some global policies such as global LRU. A user's more frequently accessed file could be evicted by a less frequently accessed file just because the first file's aggregate frequency across all users is lower than the second file. We've implemented "pluggable policy" in the system and expose a simple API for applications to pick best replacement policy.

```
Client.setCachePolicy(Policy.LRU)
Client.setCachePolicy(Policy.LFU)
Client.pinFile(fileId)
```

Currently, our implementation of FairRide supports LRU (Least-Recently-Used) and LFU (Least-Frequently-Used), as well as policies that are more suited for data-parallel analytics workloads, e.g. LIFE or LFU-F that preserves all-or-nothing properties for cached files [12]. Another feature FairRide supports is "pinned files". Through a `pinfile(fileId)` API, a user can override the replacement policy and prioritize specified files.

Delaying The key to strategy-proofness in implementing FairRide is to emulate *probabilistic blocking* by *delaying* the read of a file which a user didn't cache before. Thus the amount of wait time has to approximate the wait time as if the file is not cached, for any type of read. We implement *delaying* by simply sleeping the thread before giving a data buffer to the Tachyon client. The delay time is calculated by $\frac{size(buffer)}{BW_{disk}}$, with BW_{disk} being the pre-measured disk bandwidth on the node, and $size(buffer)$ being the size of the data buffer sent to the client. The measured bandwidth is likely an over-estimate of run-time disk bandwidth due to I/O contention when system is in operation. This causes shorter delay, higher efficiency, and less strategy-proofness, though a strategic user should gain very little from this over-estimate.

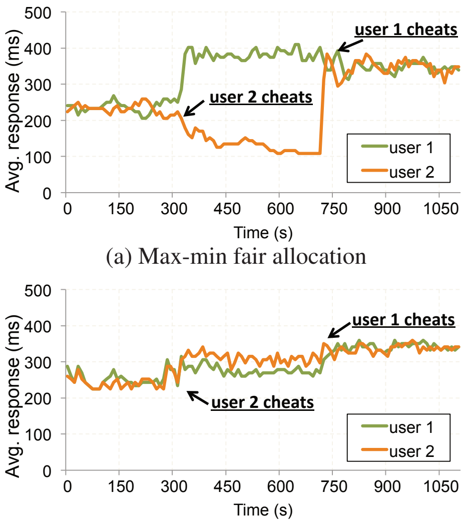
Node-based Policy Enforcement

Tachyon is a distributed system comprised of multiple worker nodes. We enforce allocation policies independently at each node. This means that data is always cached locally at the node when being read, and that when the node is full, we evict from the user who uses up most memory on that node. This scheme allows a node to select an evicting user and perform cache replacement without any global coordination.

The lack of global coordination can incur some efficiency penalty, as a user is only guaranteed to get at least $1/n$-th of memory on each node, but not necessarily $1/n$-th of total memory across the cluster. This happens when users have access skew across nodes. To give an example, suppose a cluster of two nodes, each with 40GB memory. One user has 30GB frequently accessed data on node 1 and 10GB on node 2, and another user has 10GB frequently accessed data on node 1 and 30GB on node 2. Allocating 30GB on node 1 and 10GB on node 2 to the first user will outperform a 20 to 20 even allocation on each node, in terms of hit ratio for both users. Note that such allocation is still fair globally – each user gets 40GB memory in total. Our evaluation results in Section 7.6 will show that node-based scheme is within $3\% \sim 4\%$ compared to global fairness, because of the self-balance nature of big data workloads on Tachyon.

7 Experimental Results

We evaluated FairRide using both micro- and macro-benchmarks, by running EC2 experiments on Tachyon, as well as large-scale simulations replaying production workloads. The number of users in the workloads varies from 2 to 20. We show that while non-strategy-proof policies can cause everybody worse-off by a large margin (1.9×), FairRide can prevent user starvation within



(a) Max-min fair allocation

(b) FairRide

Figure 5: Miss ratio for two users. At $t = 300s$, user 2 started cheating. At $t = 700s$, user 1 joined cheating.

4% of global efficiency. It is 2.6× better than isolated caches in terms of job time reduction, and gives 27% higer utilization compared to max-min fairness.

We start by showing how FairRide can dis-incentivize cheating users by blocking them from accessing files that they don't cache, in Section 7.1. In Section 7.2, we compare FairRide against a number of schemes, including max-min fairness using experiments on multiple workloads: TPC-H, YCSB and a HDFS production log. Section 7.3 and Section 7.4 demonstrate FairRide's benefits with multiple users and pluggable policies. Finally, in Section 7.5, we use Facebook traces that are collected from a 2000-node Hadoop cluster to evaluate the performance of FairRide in large-scale clusters.

7.1 Cheating and Blocking

In this experiment, we illustrate how FairRide can prevent a user from cheating. We ran two applications on a 5-node Amazon EC2 cluster. The cluster contains one master node and four worker nodes, each configured with 32GB memory. Each application accessed 1000 data blocks (128MB each), among which 500 were shared. File access complied with Zipf distribution. We assumed users knew a priori which files are shared, and could cheat by making excessive accesses to non-shared files. We used LRU as cache replacement policy for this experiment.

We ran the experiment under two different schemes, max-min fair allocation (Figure 5a) and FairRide (Figure 5b). Under both allocations, the two users got similar average block response time (226ms under max-min, 235ms under FairRide) at the beginning ($t < 300s$). For max-min fair allocation, when user 2 started to cheat at $t = 300s$, she managed to lower her miss ratio over time ($\sim$130ms), while user 1 got degraded performance with 380ms. At $t = 750s$, user 1 also started to cheat and both users stayed at high miss ratio (315ms). In this particular case, there was strong incentive for both the users to cheat at any point of time because cheating could always decrease the cheater's miss ratio (226ms→130ms,380ms→315ms). Unfortunately, both users get worse performance compared to not cheat all. Such a prisoner's dilemna would not happen with FairRide (Figure 5b). When user 2 cheated at $t = 300s$, her response time instead increases to 305ms . Because of this, both users would rather not cheat under FairRide and behave truthfully.

7.2 Benchmarks with Multiple Workloads

Now we evaluate FairRide by running three workloads on a EC2 cluster.

- **TPC-H** The TPC-H benchmark [11] is a set of decision support queries based on those used by retailers such as Amazon. The queries can be separated into two main groups: a sales-oriented group and a supply-oriented group. These two groups of queries have some separate tables, but also share common tables such as those maintaining inventory records. We treated two query groups as from two independent users.

- **YCSB** The Yahoo! Cloud Serving Benchmark provides a framework and common set of workloads for evaluating the performance of key-value serving stores. We implemented a YCSB client and ran multiple YCSB workloads to evaluate FairRide. We let half of files to be shared across users.

- **Production Cluster HDFS Log** The HDFS log is collected from a production Hadoop cluster at a large Internet company. It contains detailed information such as access timestamps and access user/group information. We found that more than 30% of files are shared by at least two users.

We ran each workload under the following allocation schemes: 1) *isolation*: statically partition the memory space across users; 2) *best-case*: i.e. max-min fair allocation and assume no user cheats; 3) FairRide: our solution which uses *delaying* to prevent cheating; 4) *max-min* : max-min fair allocation with half users trying to game the system. We used LRU as the default cache replacement algorithm for all users and assumed cheating users know what files are shared.

We focus on three questions: 1) does sharing the cache improve performance significantly? (comparing performance gain over *isolation*) 2) can FairRide prevent cheating with small efficiency loss? (comparing FairRide with *best-case*) 3) does cheating degrade system performance significantly? (comparing FairRide with *max-min*).

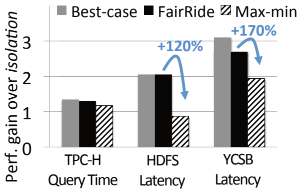


Figure 6: Summary of performance results for three workloads, showing the gain compared to isolated caches.

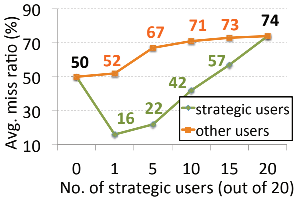


Figure 7: Average miss ratios of cheating users and non-cheating users, when there are multiple cheaters.

To answer these questions, we plot the relative gain of three schemes compared to *isolation*, as shown in Figure 6. In general, we find sharing the cache can improve performance by 1.3∼3.1×, with *best-case*. If users cheat, 15%∼220% of the gain will be lost. For the HDFS workload, we also observe that cheating causes a performance drop below *isolation*. While FairRide is very close to *best-case* with 3%∼13% overhead, it prevents the undesirable performance drop.

There are other interesting observations to note. First of all, the overhead of FairRide, is more noticeable in the YCSB benchmark and TPC-H than in the HDFS trace. We find that this is because the most shared files in the HDFS prodcution trace are among the top accessed files for both users. Therefore, both users would cache the shared files, resulting in less *blocking/delaying*. Secondly, cheating user benefits less in the HDFS trace, this is due to fact that the access distribution across files are highly long tailed in that trace, so that even cheating help user gain more memory, it doesn't show up significantly in terms of miss ratio. Finally, there is a varied degree of connection between miss ratio and application performance (read latency, query time), e.g., YCSB's read latency is directly linked to miss ratio change, while TPC-H's query response time is relatively stable. This is because, for the latter, a query typically consists of multiple stages of parallel tasks. As the completion time of a stage is decided by the slowest task, caching could only help when all tasks speed up. Therefore, a caching algorithm that can provide all-or-nothing caching for parallel tasks is needed to speed up query response time. We evaluated the Facebook trace with such a caching algorithm in Seciton 7.5.

7.3 Many Users

We want to understand how the effect of cheating relates to the number of active users in the system. In this experiment, we replay YCSB workloads with 20 users, where each pair of users have a set of shared files that they access commonly. Users can cheat by making excessive access to their private files. We increase the number of strategic users in different runs and plot the average miss ratio for both the strategic user group and the truthful user group in Figure 7. As expected, the miss ratio of the truthful group increases when there is a growing number of strategic users. What's interesting is that for the strategic group, the benefit they can exploit decreases as more and more users joining the group. With 12 strategic users, even the strategic group has worse performance compared to the no-cheater case. Eventually both groups converge at a miss ratio of 74%.

7.4 Pluggable Policies

Next, we evaluated the benefit of allowing pluggable policies. We ran three YCSB clients concurrently with each client running a different workload. The characteristics of the three workloads are summarized below:

User ID	Workload	Distribution	Replacement
1	YCSB(a)	zipfian	LFU
2	YCSB(d)	latest-most	LRU
3	YCSB(e)	scan	priority [3]

In the experiment, each YCSB client sets up the best replacement specified in the above table with the system. We compared our system with traditional caching systems that support only configuration of one uniform replacement policy, applied to all users. We ran the system with uniform configuration three times, each time with a different policy (LRU, LFU and priority). As shown in Figure 8, by allowing the users to specify a best replacement policy on their own, our system is able to provide gain of the best case for each of the user among all uniform configurations.

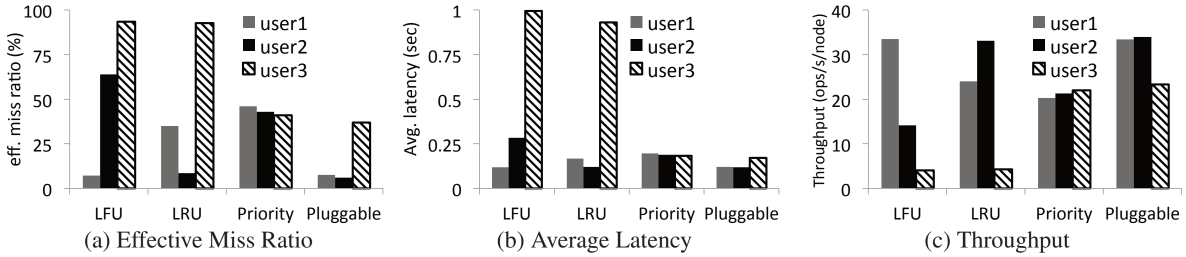
7.5 Facebook workload

Our trace-driven simulator performed a detailed and faithful replay of a task-level trace of Hadoop jobs collected from a 2000-node cluster from Facebook during the week of October 2010. Our replay preserved read and write sizes of tasks, locations of input data as well as job characteristics of failures, stragglers.

To make the effect of caching relevant to job completion time, we also use LIFE and LFU-F from PACMan [12] as cache replacement policies. These policies performed *all-or-nothing* cache replacement for files and can improve job completion time better than LRU or LFU, as it speeds all concurrent tasks in one stage [12]. In a nutshell, LIFE evicts files based on largest-incomplete-file-first eviction, and LFU-F is based on least-accessed-incomplete-file-first. We also set each

[3] Priority replacement means keeping a fixed set of files in cache. Not the best policy here, but still better than LFU and LRU for the scan workload.

[4] Effective miss ratio. For FairRide, we count a delayed access as a "fractional" miss, with the fraction equal to the blocking probability, so we can effectively compare miss ratio between FairRide and other schemes.



(a) Effective Miss Ratio

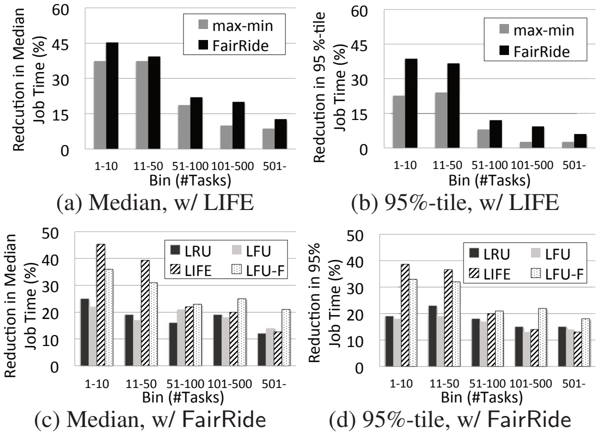
(b) Average Latency

(c) Throughput

Figure 8: Pluggable policies.

	job time		cluster eff.		eff. miss%[4]	
	u1	u2	u1	u2	u1	u2
isolation	17%	15%	23%	22%	68%	72%
global	54%	29%	55%	35%	42%	60%
best-case	42%	41%	47%	43%	48%	52%
max-min	30%	43%	35%	47%	63%	46%
FairRide	39%	40%	45%	43%	50%	55%

Table 2: Summary of simulation results on reduction in job completion time, cluster efficiency improvement and hit ratio under different scheme, with no caching as baseline.



(a) Median, w/ LIFE

(b) 95%-tile, w/ LIFE

(c) Median, w/ FairRide

(d) 95%-tile, w/ FairRide

Figure 9: Overall reduction in job completion time for Facebook trace.

node in the cluster with 20Gb of memory so miss ratio was around 50%. The conclusion would hold for a wider range of memory size.

We adopted a more advanced model of cheating in this simulation. Instead of assuming users know what files are shared a priori, a user cheats based on the cached files she observes in the cluster. For example, for a non-blocking scheme such as max-min fairness, a user can figure out what shared files are cached by other users by continuously probing the system. She would avoid sharing the cost of those files and only cache files for her own interest.

Caching improves overall performance of the system. Table 2 provides a summary of reduction in job completion time and improvement in cluster efficiency (total task run-time reduction) compared to a baseline with no caching, as well as miss ratio numbers. Similar to previous experiments, *isolation* gave lowest gains for both users and *global* improved users unevenly (compared to *best-case*). FairRide suffered minimal overhead of blocking (2% and 3% in terms of miss ratio compared to *best-case*, 4% of cluster efficiency) but could prevent cheating of user 2 that can potentially hurt user 1 by 15% in terms of miss ratio. Similar comparisons were observed in terms of job completion and cluster efficiency, FairRide can outperform *max-min* by 27% in terms of efficiency and has 2.6× more improvement over *isolation*.

Figure 9 also shows the reduction in job completion time across all users, plotted in median completion time (a) and 95 percentile completion time (b) respectively. FairRide preserved better overall reduction compared to *max-min*. This was due to the fact that marginal improvement of the cheating user was smaller than the performance drop of the cheated. FairRide also prevented cheating from greatly increasing the tail of job completion time (95 percentile) as the metric was more dominated by the slower user. We also show the improvement of FairRide under different cache policies in (c) and (d).

7.6 Comparing Global FairRide

How much performance penalty does node-based FairRide suffer compared to global FairRide, if any? To answer this question, we ran another simulation with the Facebook trace to compare against two global FairRide schemes. The two global schemes both select a evicting user based on users' global usage, but differ in how they pick evicting blocks: a "naive" global scheme chooses from only blocks on that node, similar to the node-based approach, and an "optimized" global scheme chooses from any user blocks in the cluster. We use LIFE as the replacement policy for both users.

Cluster size	200	500	1000
Node-based FairRide	51%	44%	41%
Global FairRide, Naive	25%	21%	17%
Global FairRide, Optimized	54%	47%	44%

Table 3: Comparing against global schemes. Keep total memory size as constant while varying the number of nodes in the cluster. Showing improvement over no cache as in the reduction in median job completion time.

As we find out, the naive global scheme has a great performance drop (23%∼25% improvement difference compared to node-based FairRide), noticeably in Table 3. This is due to the fact that the naive scheme is unable to allocate in favor of frequently accessing user per node. With the naive global scheme, memory allocations on each node quickly stabilizes based on initial user accesses. A user can get an unnecessarily large portion of memory on a node because she accesses data earlier than the other, although her access frequency on that node is low in general. The optimized global scheme fixes this issue by allowing a user to evict least preferred data in the whole cluster and it makes sure the $1/n$-th of memory allocated must store her most preferred data. We observe an increase of average hit ratio by 24% with the optimized scheme, which reflects the access skew for the underlying data. What's interesting is that the optimized global scheme is only 3%∼4% better than node-based scheme in terms of job complete time improvement. In addition to the fact data skew is not huge (considering 24% increase for hit ratio), the all-or-nothing property of data-parallel caching again comes into play. Global scheme on average increases the number of completely cached files by only 7%, and because now memory allocation is skewed across the cluster, there is an increased chance that tasks cannot be scheduled to co-locate with cached data, due to CPU slot limitation. Finally, we also observe that as the number of nodes increases (while keeping the total CPU slots and memory constant), there is a decrease in improvement in all three schemes, due to less tasks can be scheduled with cache locality.

8 Related Works

Management of shared resources has always been been an important subject. Over the past decades, researchers and practitioners have considered the sharing of CPU [39, 15, 35, 40] and network bandwidth [28, 13, 17, 23, 33, 36], and developed a plethora of solutions to allocate and schedule these resources. The problem of cache allocation for better isolation, quality-of-service [24] or attack resilience [29] has also been studied under various contexts, including CPU cache [25], disk cache [31] and caching in storage systems [34].

One of the most popular allocation policies is *fair sharing* [16] or *max-min fairness* [27, 14]. Due to the nice properties, it has been implemented using various methods, such as round-robin, proportional resource sharing [39] and fair queuing [18], and has been extended to support multiple resource types [20] and resource constraints [21]. The key differentiator for our work from the ones mentioned above, is that we consider shared data. None of the works above identifies the impossibility of three important properties with shared files.

There are other techniques that have been studied to provide fairness and efficiency of shared cache. Prefetching of data into the cache before access, either through hints from applications [31] or predication [22], can improve the overall system efficiency. Profiling applications [25] is useful for provding application-sepcific information. We view these techniques as orthogonal to our work. Other techniques such as throttling access rate requires the system to identify good thresholds.

9 Conclusions

In this paper, we study the problem of cache allocation in a multi-user environment. We show that with data sharing, it is not possible to find an allocation policy that achieves isolation-guarantee, strategy-proofness and Pareto-efficiency simultaneously. We propose a new policy called FairRide. Unlike previous policies, FairRide provides both isolation-guarantee (so a user gets better performance than on isolated cache) and strategy-proofness (so users are not incentivized to cheat), by blocking access from cheating users. We provide an efficient implementation of the FairRide system and show that in many realistic workloads, FairRide can outperform previous policies when users cheat. The two nice properties of FairRide come at the cost of Pareto-efficiency. We also show that FairRide's cost is within 4% of total efficiency in some of the production workloads, when we conservatively assume users don't cheat. Based of the appealing properties and relatively small overhead, we believe that FairRide can be a practical policy for real-world cloud environments.

Acknowledgement

We thank Radhika Mittal, Shivaram Venkataraman, Rachit Agarwal and other members of Berkeley CS and AMPLab for providing many feedbacks on this work; Scott Shenker and Eric J. Friedman for providing guidance during early stage of the project; Srikanth Kandula for helpful feedbacks on the draft; and the anonymous reviewers for valuable comments; and finally, our shepherd, Thomas Moscibroda, for helping to shape the final version of the paper. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, Guavus, HP, Huawei, Intel, Microsoft, Pivotal, Samsung, Schlumberger, Splunk, State Farm, Virdata and VMware.

References

[1] Amazon elasticache. https://aws.amazon.com/elasticache/.

[2] Apache Hadoop. http://hadoop.apache.org/.

[3] Azure cache - redis cache cloud service. http://azure.microsoft.com/en-us/services/cache/.

[4] Distributed Memory: Supporting Memory Stor-

age in HDFS. http://hortonworks.com/blog/ddm/.

[5] Hdfs caching. http://blog.cloudera.com/blog/2014/08/new-in-cdh-5-1-hdfs-read-caching/.

[6] Isolation in Memcached or Redis. http://goo.gl/FYfrOK;http://goo.gl/iocFrt;http://goo.gl/VeJHvs.

[7] Memcached, a distributed memory object caching system. http://memcached.org/.

[8] MemSQL In-Memory Database. http://http://www.memsql.com/.

[9] Redis. http://redis.io/.

[10] The column-store pioneer: MonetDB. https://www.monetdb.org.

[11] TPC-H. http://www.tpc.org/tpch.

[12] Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. Pacman: coordinated memory caching for parallel jobs. In *NSDI'12*.

[13] Bennett, J. C., and Zhang, H. Wf2q: worst-case fair weighted fair queueing. In *INFOCOM'96*.

[14] Cao, Z., and Zegura, E. W. Utility max-min: An application-oriented bandwidth allocation scheme. In *INFOCOM'99*.

[15] Caprita, B., Chan, W. C., Nieh, J., Stein, C., and Zheng, H. Group ratio round-robin: O (1) proportional share scheduling for uniprocessor and multiprocessor systems. In *ATC'05*.

[16] Crowcroft, J., and Oechslin, P. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. *SIGCOMM CCR, 1998*.

[17] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM CCR, 1989*.

[18] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM'89*.

[19] Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W. Sap hana database: Data management for modern business applications. *SIGMOD Rec., 2012*.

[20] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant resource fairness: Fair allocation of multiple resource types. NSDI'11.

[21] Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Choosy: Max-min fair sharing for datacenter jobs with constraints. EuroSys'13.

[22] Gill, B. S., and Bathen, L. A. D. Amp: Adaptive multi-stream prefetching in a shared cache. In *FAST'07*.

[23] Goyal, P., Vin, H. M., and Chen, H. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *SIGCOMM CCR, 1996*.

[24] Iyer, R., Zhao, L., Guo, F., Illikkal, R., Makineni, S., Newell, D., Solihin, Y., Hsu, L., and Reinhardt, S. Qos policies and architecture for cache/memory in cmp platforms. In *SIGMETRICS'07*.

[25] Kim, S., Chandra, D., and Solihin, Y. Fair cache sharing and partitioning in a chip multiprocessor architecture. PACT'04.

[26] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SOCC'14*.

[27] Ma, Q., Steenkiste, P., and Zhang, H. Routing high-bandwidth traffic in max-min fair share networks. In *SIGCOMM CCR, 1996*.

[28] Massoulié, L., and Roberts, J. Bandwidth sharing: objectives and algorithms. In *INFOCOM'99*.

[29] Moscibroda, T., and Mutlu, O. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security'07*.

[30] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS OSR, 2010*.

[31] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed prefetching and caching. *SIGOPS'95*.

[32] Piatek, M., Isdal, T., Anderson, T., Krishnamurthy, A., and Venkataramani, A. Do incentives build robustness in bit torrent. In *NSDI'07*.

[33] Shreedhar, M., and Varghese, G. Efficient fair queuing using deficit round-robin. *TON'96*.

[34] Shue, D., Freedman, M. J., and Shaikh, A. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI'12*.

[35] Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S. K., Gehrke, J. E., and Plaxton, C. G. A proportional share resource allocation algorithm for real-time, time-shared systems. In *RTSS'96*.

[36] Stoica, I., Shenker, S., and Zhang, H. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *SIGCOMM'98*.

[37] Stonebraker, M., and Weisberg, A. The

voltdb main memory dbms. http://voltdb.com/.

[38] Verma, A., Pedrosa, L. D., Korupolu, M., Oppenheimer, D., and Wilkes, J. Large scale cluster management at google with borg. Eurosys'15.

[39] Waldspurger, C. A., and Weihl, W. E. Lottery scheduling: Flexible proportional-share resource management. In *OSDI'94*.

[40] Waldspurger, C. A., and Weihl, W. E. Stride scheduling: deterministic proportional-share resource management. In *MIT Tech Report, 1995*.

[41] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12*.

HUG: Multi-Resource Fairness for Correlated and Elastic Demands

Mosharaf Chowdhury[1], Zhenhua Liu[2], Ali Ghodsi[3], Ion Stoica[3]

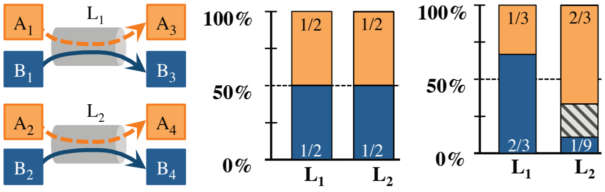
[1]*University of Michigan* [2]*Stony Brook University* [3]*UC Berkeley, Databricks Inc.*

Abstract

In this paper, we study how to optimally provide isolation guarantees in multi-resource environments, such as public clouds, where a tenant's demands on different resources (links) are *correlated*. Unlike prior work such as Dominant Resource Fairness (DRF) that assumes static and fixed demands, we consider *elastic* demands. Our approach generalizes canonical max-min fairness to the multi-resource setting with correlated demands, and extends DRF to elastic demands. We consider two natural optimization objectives: isolation guarantee from a tenant's viewpoint and system utilization (work conservation) from an operator's perspective. We prove that in non-cooperative environments like public cloud networks, there is a strong tradeoff between optimal isolation guarantee and work conservation when demands are elastic. Even worse, work conservation can even decrease network utilization instead of improving it when demands are inelastic. We identify the root cause behind the tradeoff and present a provably optimal allocation algorithm, *High Utilization with Guarantees* (HUG), to achieve maximum attainable network utilization without sacrificing the optimal isolation guarantee, strategy-proofness, and other useful properties of DRF. In cooperative environments like private datacenter networks, HUG achieves both the optimal isolation guarantee and work conservation. Analyses, simulations, and experiments show that HUG provides better isolation guarantees, higher system utilization, and better tenant-level performance than its counterparts.

1 Introduction

In shared, multi-tenant environments such as public clouds [2, 5, 6, 8, 40], the need for predictability and the means to achieve it remain a constant source of discussion [15, 44, 45, 50, 51, 58, 59, 61, 62]. The general consensus – recently summarized by Mogul and Popa [53] – is that tenants expect guaranteed minimum bandwidth (i.e., *isolation guarantee*) for performance predictability, while network operators strive for *work conservation* to achieve high utilization and *strategy-proofness* to ensure isolation.



(a) **(b)** Max-min fair. **(c)** DRF [33]

Figure 1: Bandwidth allocations in two independent links (a) for tenant-A (orange) with correlation vector $\overrightarrow{d_A} = \langle \frac{1}{2}, 1 \rangle$ and tenant-B (dark blue) with $\overrightarrow{d_B} = \langle 1, \frac{1}{6} \rangle$. Shaded portions are not allocated to any tenant.

Max-min fairness [43] – a widely-used [16, 25, 34, 35, 55, 63, 64] allocation policy – achieves all three in the context of a single link. It provides the *optimal* isolation guarantee by maximizing the minimum amount of bandwidth allocated to each flow. The bandwidth allocation of a user (tenant) determines her progress – i.e., how fast she can complete her data transfer. It is work-conserving, because, given enough demand, it allocates the entire bandwidth of the link. Finally, it is strategyproof, because tenants cannot get more bandwidth by lying about their demands (e.g., by sending more traffic).

However, a datacenter network involves many links, and tenants' demands on different links are often *correlated*. Informally, we say that the demands of a tenant on two links i and j are correlated, if for every bit the tenant sends on link-i, she sends at least α bits on link-j. More formally, with every tenant-k, we associate a correlation vector $\overrightarrow{d_k} = \langle d_k^1, d_k^2, \ldots, d_k^n \rangle$, where $d_k^i \leq 1$, which captures the fact that for every d_k^i bits tenant-k sends on link-i, it should send at least d_k^j bits on link-j.

Examples of applications with *correlated demands* include optimal shuffle schedules [22, 23], long-running services [19, 52], multi-tiered enterprise applications [39], and realtime streaming applications [10, 69]. Consider the example in Figure 1a with two independent links and two tenants. The correlation vector $\overrightarrow{d_A} = \langle \frac{1}{2}, 1 \rangle$ means that (i) link-2 is tenant-A's bottleneck, (ii) for every $\mathcal{M}_A$ rate tenant-A is allocated on the bottleneck link, she requires *at least* $\mathcal{M}_A/2$ rate on link-1, resulting in a progress of $\mathcal{M}_A$, and (iii) except for the bottleneck link, tenants' demands are elastic, meaning tenant-A can use more than $\mathcal{M}_A/2$ rate on link-1.[1] Similarly, tenant-B requires *at least* $\mathcal{M}_B/6$ on link-2 for $\mathcal{M}_B$ on link-1. If we denote the rate allocated to tenant-k on link-i by a_k^i, then $\mathcal{M}_k = \min_i \left\{ \frac{a_k^i}{d_k^i} \right\}$, the minimum demand-normalized rate allocation over all links, captures her progress.

In this paper, we want *to generalize max-min fairness to tenants with correlated and elastic demands while maintaining its desirable properties: optimal isolation guarantee, high utilization, and strategy-proofness.*

[1] While it does not improve the instantaneous progress of tenant-A, it increases network utilization, which is desired by the operators.

Intuitively, we want to maximize the minimum progress over all tenants, i.e., **maximize** $\min_k \mathcal{M}_k$, where $\min_k \mathcal{M}_k$ corresponds to the isolation guarantee of an allocation algorithm. We make three observations. First, when there is a single link in the system, this model trivially reduces to max-min fairness. Second, getting more aggregate bandwidth is not always better. For tenant-A in the example, $\langle 50\text{Mbps}, 100\text{Mbps}\rangle$ is better than $\langle 90\text{Mbps}, 90\text{Mbps}\rangle$ or $\langle 25\text{Mbps}, 200\text{Mbps}\rangle$, even though the latter ones have more bandwidth in total. Third, simply applying max-min fairness to individual links is not enough. In our example, max-min fairness allocates equal resources to both tenants on both links, resulting in allocations $\langle \frac{1}{2}, \frac{1}{2}\rangle$ on both links (Figure 1b). Corresponding progress ($\mathcal{M}_A = \mathcal{M}_B = \frac{1}{2}$) result in a suboptimal isolation guarantee ($\min\{\mathcal{M}_A, \mathcal{M}_B\} = \frac{1}{2}$).

Dominant Resource Fairness (DRF) [33] extends max-min fairness to multiple resources and prevents such sub-optimality. It equalizes the shares of dominant resources – link-2 (link-1) for tenant-A (tenant-B) – across *all* tenants with correlated demands and maximizes the isolation guarantee in a strategyproof manner. As shown in Figure 1c, using DRF, both tenants have the *same* progress – $\mathcal{M}_A = \mathcal{M}_B = \frac{2}{3}$, 50% higher than using max-min fairness on individual links. Moreover, DRF's isolation guarantee ($\min\{\mathcal{M}_A, \mathcal{M}_B\} = \frac{2}{3}$) is optimal across all possible allocations and is strategyproof.

However, DRF assumes *inelastic* demands [40], and it is not work-conserving. For example, bandwidth on link-2 in shades is not allocated to either tenant. In fact, we show that DRF can result in *arbitrarily low utilization* (Lemma 6). This is wasteful, because unused bandwidth cannot be recovered.

We start by showing that strategy-proofness is a *necessary* condition for providing the optimal isolation guarantee – i.e., to **maximize** $\min_k \mathcal{M}_k$ – in non-cooperative environments (§2). Next, we prove that work conservation – i.e., when tenants are allowed to use unallocated resources, such as the shaded area in Figure 1c, without constraints – spurs a race to the bottom. It incentivizes each tenant to continuously lie about her demand correlations, and in the process, it decreases the amount of useful work done by all tenants! Meaning, simply making DRF work-conserving can do more harm than good.

We propose a two-stage algorithm, *High Utilization with Guarantees* (HUG), to achieve our goals (§3). Figure 2 surveys the design space for cloud network sharing and places HUG in context by following the thick lines. At the highest level, unlike many alternatives [13, 14, 37, 44], HUG is a dynamic allocation algorithm. Next, HUG enforces its allocations at the tenant-/network-level, because flow- or (virtual) machine-level allocations [61, 62] do not provide isolation guarantee.

Due to the hard tradeoff between optimal isolation guarantee and work conservation in non-cooperative environments, *HUG ensures the highest utilization possible while maintaining the optimal isolation guarantee*. It incentivizes tenants to expose their true demands, ensuring that they actually consume their allocations instead of causing collateral damages. In cooperative environments, where strategy-proofness might be a non-requirement, *HUG simultaneously ensures both work conservation and the optimal isolation guarantee*. In contrast, existing solutions [33, 45, 51, 58, 59] are sub-optimal in both environments. Overall, HUG generalizes single- [25, 43, 55] and multi-resource max-min fairness [27, 33, 38, 56] and multi-tenant network sharing solutions [45, 51, 58, 59, 61, 62] under a unifying framework.

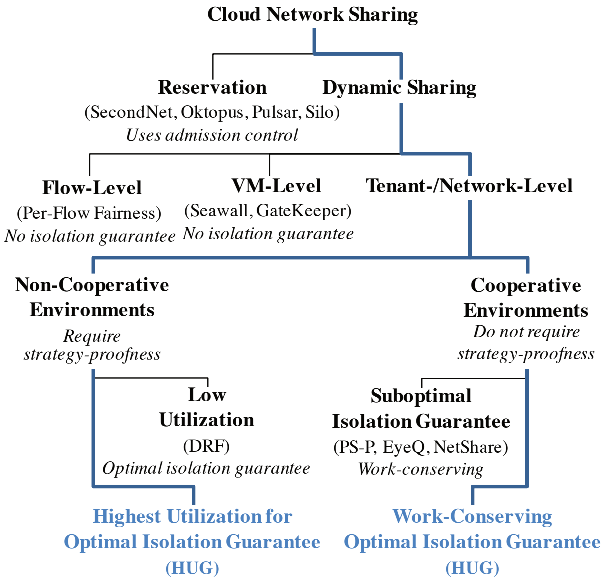


Figure 2: Design space for cloud network sharing.

HUG is easy to implement and scales well. Even with $100,000$ machines, new allocations can be centrally calculated and distributed throughout the network in less than a second – faster than that suggested in the literature [13]. Moreover, each machine can locally enforce HUG-calculated allocations using existing traffic control tools without any changes to the network (§4).

We demonstrate the effectiveness of our proposal using EC2 experiments and trace-driven simulations (§5). In non-cooperative environments, HUG provides the optimal isolation guarantee, which is $7.4\times$ higher than existing network sharing solutions like PS-P [45, 58, 59] and $7000\times$ higher than traditional per-flow fairness, and $1.4\times$ better utilization than DRF for production traces. In cooperative environments, HUG outperforms PS-P and per-flow fairness by $1.48\times$ and $17.35\times$ in terms of the 95th percentile slowdown of job communication stages, and 70% jobs experience lower slowdown w.r.t. DRF.

We discuss current limitations and future research in Section 6 and compare HUG to related work in Section 7.

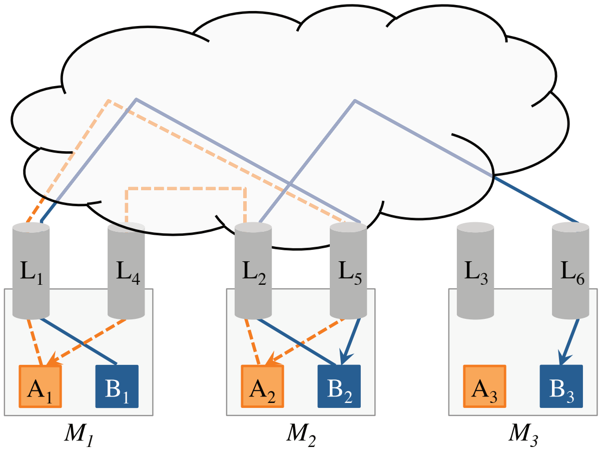


Figure 3: Two VMs from tenant-A (orange) and three from tenant-B (dark blue) and their communication patterns over a 3×3 datacenter fabric. The network fabric has three uplinks (L_1–L_3) and three downlinks (L_4–L_6) corresponding to the three physical machines.

2 Motivation

In this section, we elaborate on the assumptions and notations used in this paper and summarize the three desirable requirements – optimal isolation guarantee, high utilization, proportionality – for bandwidth allocation across multiple tenants. Later, we show the tradeoff between optimal isolation guarantee and high utilization, identifying work conservation as the root cause.

2.1 Background

We consider Infrastructure-as-a-Service (e.g., EC2 [2], Azure [8], and Google Compute [5]) and Container-as-a-Service (e.g., Mesos [40] and Kubernetes [6]) models where tenants pay per-hour flat rates for virtual machines (VMs) and containers.[2]

We abstract out the datacenter network as a non-blocking switch (i.e., the fabric/hose model [11, 12, 14, 23, 28, 45, 49]) with P physical machines connected to it. Each machine has full-duplex links (i.e., $2P$ independent links) and can host one or more VMs from different tenants. Figure 3 shows an example. We assume that VM placement and routing are implemented independently. Not only does this model provide analytical simplicity, it is mostly a reality today: recent EC2 and Google datacenters have full bisection bandwidth networks [4, 7].

We denote the correlation vector of the k-th tenant ($k \in \{1, ..., M\}$) as $\overrightarrow{d_k} = \langle d_k^1, d_k^2, \ldots d_k^{2P} \rangle$, where d_k^i and d_k^{P+i} ($1 \leq i \leq P$) respectively denote the uplink and downlink demands normalized[3] by link capacities (C^i) and $\sum_{i=1}^{P} d_k^i = \sum_{i=1}^{P} d_k^{P+i}$.

$\overrightarrow{d_k}$	Correlation vector of tenant-k's demand
$\overrightarrow{a_k}$	Guaranteed allocation to tenant-k
$\mathcal{M}_k$	Progress of tenant-k; $\mathcal{M}_k := \min\limits_{1 \leq i \leq 2P} \left\{ \frac{a_k^i}{d_k^i} \right\}$, where subscript i stands for link-i
$\overrightarrow{c_k}$	Actual resource consumption of tenant-k

Isolation Guarantee	$\min_k \mathcal{M}_k$
Optimal Isolation Guarantee	$\max \{\min_k \mathcal{M}_k\}$
(Network) Utilization	$\sum_i \sum_k c_k^i$

Table 1: Important notations and definitions.

For the example in Figure 3, consider tenant correlation vectors:

$$\overrightarrow{d_A} = \langle \frac{1}{2}, 1, 0, 1, \frac{1}{2}, 0 \rangle$$
$$\overrightarrow{d_B} = \langle 1, \frac{1}{6}, 0, 0, 1, \frac{1}{6} \rangle$$

where $d_k^i = 0$ indicates the absence of a VM and $d_k^i = 1$ indicates the bottleneck link(s) of a tenant.

Correlation vectors depend on tenant applications, that can range from *elastic-demand* batch jobs [3, 24, 42, 68] to long-running services [19, 52], multi-tiered enterprise applications [39], and realtime streaming applications [9, 69] with *inelastic demands*. We focus on scenarios where a tenant's demand changes at the timescale of seconds or longer [13, 18, 58], and she can use provider-allocated resources in any way for her own workloads.

2.2 Inter-Tenant Network Sharing Requirements

Given correlation vectors of M tenants, a cloud provider must use an allocation algorithm $\mathcal{A}$ to determine the allocations of each tenant:

$$\mathcal{A}(\{\overrightarrow{d_1}, \overrightarrow{d_2}, \ldots, \overrightarrow{d_M}\}) = \{\overrightarrow{a_1}, \overrightarrow{a_2}, \ldots, \overrightarrow{a_M}\}$$

where $\overrightarrow{a_k} = \langle a_k^1, a_k^2, \ldots a_k^{2P} \rangle$ and a_k^i is the fraction of link-i guaranteed to the k-th tenant.

As identified in previous work [15, 58], any allocation policy $\mathcal{A}$ must meet three requirements – (optimal) isolation guarantee, high utilization, and proportionality – to fairly share the cloud network:

1. **Isolation Guarantee:** VMs should receive minimum bandwidth guarantees *proportional* to their correlation vectors so that tenants can estimate *worst-case* performance. Formally, *progress* of tenant-k ($\mathcal{M}_k$) is defined as her minimum demand satisfaction ratio across the entire fabric:

$$\mathcal{M}_k = \min_{1 \leq i \leq 2P} \left\{ \frac{a_k^i}{d_k^i} \right\}$$

[2] We use the terms VM and container interchangeably in this paper because they are similar from the network's perspective.

[3] Normalization helps us consider heterogeneous capacities. By default we normalize the correlation vector such that the largest component equals to 1 unless otherwise specified.

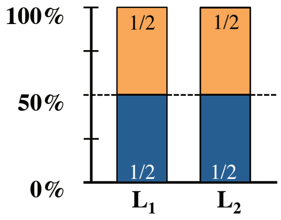


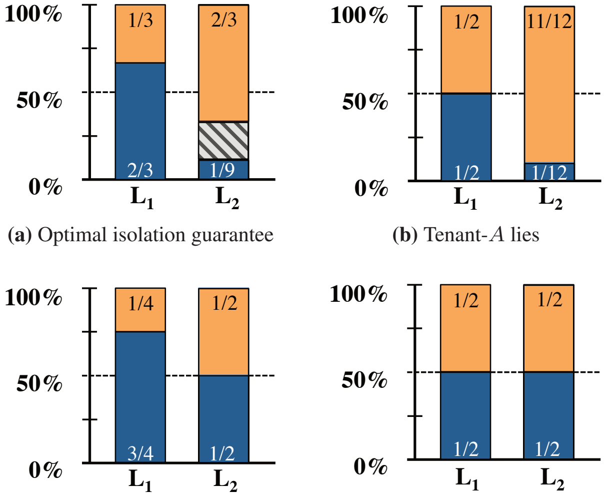
Figure 4: Bandwidth consumptions of tenant-A (orange) and tenant-B (dark blue) with correlation vectors $\overrightarrow{d_A} = \langle \frac{1}{2}, 1 \rangle$ and $\overrightarrow{d_B} = \langle 1, \frac{1}{6} \rangle$ using PS-P [45, 58, 59]. Both tenants run elastic-demand applications.

For example, progress of tenants A and B in Figure 4 are $\mathcal{M}_A = \mathcal{M}_B = \frac{1}{2}$.[4] Note that $\mathcal{M}_k = \frac{1}{M}$ if $\overrightarrow{d_k} = \langle 1, 1, \ldots, 1 \rangle$ for all tenants (generalizing PS-P [58]), and $\mathcal{M}_k = \frac{1}{M}$ for flows on a single link (generalizing per-flow max-min fairness [43]).
Isolation guarantee is defined as the lowest progress across all tenants, i.e., $\min\limits_k \mathcal{M}_k$.

2. **High Utilization:** Spare network capacities should be utilized by tenants with elastic demands to ensure high utilization as long as it does not decrease anyone's progress.
A related concept is *work conservation*, which ensures that either a link is fully utilized or demands from all flows traversing the link have been satisfied [43, 58]. Although existing research conflates the two [14, 15, 45, 51, 58, 59, 61, 62, 67], we show in the next section why that is not the case.
3. **Proportionality:** A tenant's bandwidth allocation should be proportional to its payment similar to resources like CPU and memory. We discuss this requirement in more details in Section 3.3.1.

2.3 Challenges and Inefficiencies of Existing Allocation Policies

Prior work also identified two tradeoffs: *isolation guarantee vs. proportionality* and *high utilization vs. proportionality*. However, it has been implicitly assumed that tenant-level *optimal* isolation guarantee[5] and network-level work conservation can coexist. Although optimal isolation guarantee and network-level work conservation can coexist for a single link – max-min fairness is an example – *optimal isolation guarantee and work conservation can be at odds when we consider the network as a whole.* This has several implications on both isolation guarantee and network utilization. In particular, we can (1) either optimize utilization, *then* maximize the isolation guarantee with best effort; or (2) optimize the isolation guarantee, *then* maximize utilization with best effort.[6] Please refer to Appendix C for more details.



(a) Optimal isolation guarantee

(b) Tenant-A lies

(c) Tenant-B lies

(d) Both lie

Figure 5: Bandwidth consumptions of tenant-A (orange) and tenant-B (dark blue) with correlation vectors $\overrightarrow{d_A} = \langle \frac{1}{2}, 1 \rangle$ and $\overrightarrow{d_B} = \langle 1, \frac{1}{6} \rangle$, when both run elastic-demand applications. (a) Optimal isolation guarantee in the absence of work conservation. With work conservation, (b) tenant-A increases her progress at the expense of tenant-B, and (c) tenant-B can do the same, which results in (d) a prisoner's dilemma.

2.3.1 Full Utilization but Suboptimal Isolation Guarantee

As shown in prior work [58, Section 2.5], flow-level and VM-level mechanisms – e.g., per-flow, per source-destination pair [58], and per-endpoint fairness [61, 62] – can easily be manipulated by creating more flows or by using denser communication patterns. To avoid such manipulations, many allocation mechanisms [45, 58, 59] equally divide link capacities at the tenant level and allow work conservation for tenants with unmet demands. Figure 4 shows an allocation using PS-P [58] with isolation guarantee $\frac{1}{2}$. If both tenants have elastic-demand applications, they will consume entire allocations; i.e., $\overrightarrow{c_A} = \overrightarrow{c_B} = \overrightarrow{a_A} = \overrightarrow{a_B} = \langle \frac{1}{2}, \frac{1}{2} \rangle$, where $\overrightarrow{c_k} = \langle c_k^1, c_k^2, \ldots c_k^{2P} \rangle$ and c_k^i is the fraction of link-i *consumed* by tenant-k. Recall that a_k^i is the guaranteed allocation of link-i to tenant-k.

However, PS-P and similar mechanisms are also suboptimal. For the ongoing example, Figure 5a shows the optimal isolation guarantee of $\frac{2}{3}$, which is higher than that provided by PS-P. In short, full utilization does not necessarily imply optimal isolation guarantee!

[4] We are continuing the example in Figure 3 but omitted the rest of $\overrightarrow{a_k}$, because there is either no contention or they are symmetric.

[5] Optimality means that the allocation maximizes the isolation guarantee across all tenants, i.e., **maximize** $\left\{ \min\limits_k \mathcal{M}_k \right\}$.

[6] Maximizing a combination of these two is also an interesting future direction.

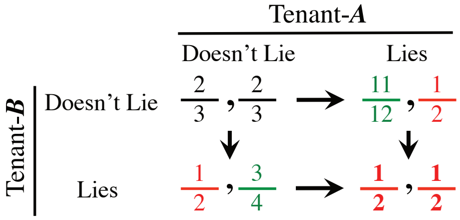


Figure 6: Payoff matrix for the example in Section 2.3. Each cell shows progress of tenant-A and tenant-B.

2.3.2 Optimal Isolation Guarantee but Low Utilization

In contrast, optimal isolation guarantee does not necessarily mean full utilization. In general, optimal isolation guarantees can be calculated using DRF [33], which generalizes max-min fairness to multiple resources. In the example of Figure 5a, each uplink and downlink of the fabric is an independent resource – $2P$ in total.

Given this premise, it seems promising and straightforward to keep the DRF-component for optimal isolation guarantee and strategy-proofness and try to ensure full utilization by allocating *all* remaining resources.In the following two subsections, we show that work conservation may render isolation guarantee no longer optimal, and even worse, may reduce useful network utilization.

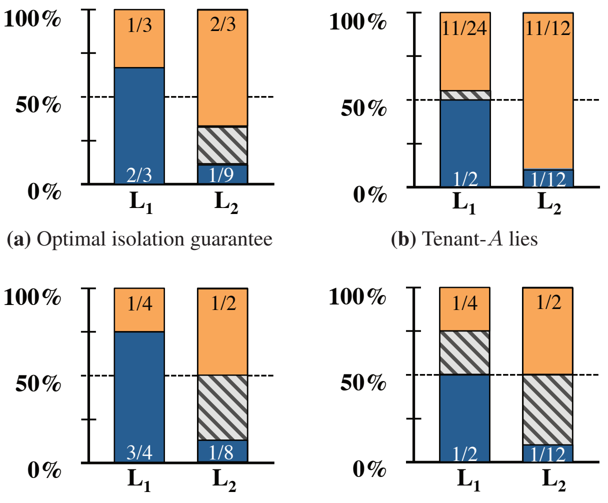
2.3.3 Naive Work Conservation Reduces Optimal Isolation Guarantee

We first illustrate that even the optimal isolation guarantee allocation degenerates into the classic prisoner's dilemma problem [30] in the presence of work conservation. In particular, we show that reporting a false correlation vector $\langle 1, 1\rangle$ is the dominant strategy for each tenant, i.e., her best option, no matter whether the other tenants tell the truth or not. As a consequence, optimal isolation guarantees decrease (Figure 6).

If tenant-A can use the spare bandwidth in link-2, she can increase her progress at the expense of tenant-B by changing her correlation vector to $\overrightarrow{d'_A} = \langle 1, 1\rangle$. With an unmodified $\overrightarrow{d_B} = \langle 1, \frac{1}{6}\rangle$, the new allocation would be $\overrightarrow{a_A} = \langle \frac{1}{2}, \frac{1}{2}\rangle$ and $\overrightarrow{a_B} = \langle \frac{1}{2}, \frac{1}{12}\rangle$. However, work conservation would increase it to $\overrightarrow{a_A} = \langle \frac{1}{2}, \frac{11}{12}\rangle$ (Figure 5b). Overall, progress of tenant-A would increase to $\frac{11}{12}$, while decreasing it to $\frac{1}{2}$ for tenant-B. As a result, the isolation guarantee decreases from $\frac{2}{3}$ to $\frac{1}{2}$.

The same is true for tenant-B as well. Consider again that only tenant-B reports a falsified correlation vector $\overrightarrow{d'_B} = \langle 1, 1\rangle$ to receive a favorable allocation: $\overrightarrow{a_A} = \langle \frac{1}{4}, \frac{1}{2}\rangle$ and $\overrightarrow{a_B} = \langle \frac{1}{2}, \frac{1}{2}\rangle$. Work conservation would increase it to $\overrightarrow{a_B} = \langle \frac{3}{4}, \frac{1}{2}\rangle$ (Figure 5c). Overall, progress of tenant-B would increase to $\frac{3}{4}$, while decreasing it to $\frac{1}{2}$ for tenant-A, resulting in the same suboptimal isolation guarantee $\frac{1}{2}$.

Since both tenants gain by lying, they would both simultaneously lie: $\overrightarrow{d'_A} = \overrightarrow{d'_B} = \langle 1, 1\rangle$, resulting in a lower isolation guarantee $\frac{1}{2}$ (Figure 5d). Both are worse off!



(a) Optimal isolation guarantee

(b) Tenant-A lies

(c) Tenant-B lies

(d) Both lie

Figure 7: Bandwidth consumptions of tenant-A (orange) and tenant-B (dark blue) with correlation vectors $\overrightarrow{d_A} = \langle \frac{1}{2}, 1\rangle$ and $\overrightarrow{d_B} = \langle 1, \frac{1}{6}\rangle$, when neither runs elastic-demand applications. (a) Optimal isolation guarantee allocation is not work-conserving. With work conservation, (b) utilization can increase or (c) decrease, based on which tenant lies. (d) However, ultimately it lowers utilization. Shaded means unallocated.

In this example, the inefficiency arises due to allocating all spare resources to the tenant who demands more. We show in Appendix B that intuitive allocation policies of all spare resources – e.g., allocating all to who demands the least, allocating equally to all tenants with non-zero demands, and allocating proportionally to tenants' demands – do not work as well.

2.3.4 Naive Work Conservation can Even Decrease Utilization

Now consider that *neither* tenant has elastic-demand applications; i.e., they can only consume bandwidth proportional to their correlation vectors. A similar prisoner's dilemma unfolds (Figure 6), but this time, network utilization decreases as well.

Given the optimal isolation guarantee allocation, $\overrightarrow{a_A} = \overrightarrow{c_A} = \langle \frac{1}{3}, \frac{2}{3}\rangle$ and $\overrightarrow{a_B} = \overrightarrow{c_B} = \langle \frac{2}{3}, \frac{1}{9}\rangle$, both tenants have the same optimal isolation guarantee: $\frac{2}{3}$, and $\frac{2}{9}$-th of link-2 remain unused (Figure 7a). One would expect work conservation to utilize this spare capacity.

Same as before, if tenant-A changes her correlation vector to $d'_A = \langle 1, 1\rangle$, she can receive an allocation $\overrightarrow{a_A} = \langle \frac{1}{2}, \frac{11}{12}\rangle$ and consume $\overrightarrow{c_A} = \langle \frac{11}{24}, \frac{11}{12}\rangle$. This increases her isolation guarantee to $\frac{11}{12}$ and total network utilization increases (Figure 7b).

Similarly, tenant-B can receive an allocation $\overrightarrow{a_B} = \langle \frac{3}{4}, \frac{1}{2}\rangle$ and consume $\overrightarrow{c_B} = \langle \frac{3}{4}, \frac{1}{8}\rangle$ to increase her isola-

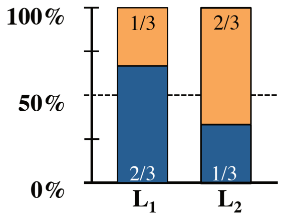


Figure 8: Optimal allocations with maximum achievable utilizations and maximum isolation guarantees for tenant-A (orange) and tenant-B (dark blue).

tion guarantee to $\frac{3}{4}$. Utilization decreases (Figure 7c).

Consequently, both tenants lie and consume $\overrightarrow{c_A} = \langle \frac{1}{4}, \frac{1}{2} \rangle$ and $\overrightarrow{c_B} = \langle \frac{1}{2}, \frac{1}{12} \rangle$ (Figure 7d). Instead of increasing, work conservation decreases network utilization!

2.4 Summary

The primary takeaways of this section are the following:

- Existing mechanisms provide either suboptimal isolation guarantees or low network utilization.
- There exists a strong tradeoff between optimal isolation guarantee and high utilization in a multi-tenant network. The key lies in strategy-proofness: optimal isolation guarantee requires it, while work conservation nullifies it. We provide a formal result about this (**Corollary 2**) in the next section.
- Unlike single links, work conservation can decrease network utilization instead of increasing it.

3 HUG: Analytical Results

In this section, we show that despite the tradeoff between optimal isolation guarantee and work conservation, it is possible to increase utilization to some extent. Moreover, we present HUG, the optimal algorithm to ensure maximum achievable utilization *without* sacrificing optimal isolation guarantees and strategy-proofness of DRF.

We defer the proofs from this section to Appendix A.

3.1 Root Cause Behind the Tradeoff: Unrestricted Sharing of Spare Resources

Going back to Figure 5, both tenants were incentivized to lie because they were receiving spare resources without any restriction due to the pursuit of work conservation.

After tenant-A lied in Figure 5b, both $\mathcal{M}_A$ and $\mathcal{M}_B$ decreased to $\frac{1}{2}$. However, by cheating, tenant-A managed to increase her allocation in link-1 to $\frac{1}{2}$ from $\frac{1}{3}$. Next, indiscriminate work conservation increased her allocation in link-2 to $\frac{11}{12}$ from the initial $\frac{1}{2}$, effectively increasing $\mathcal{M}_A$ to $\frac{11}{12}$. Similarly in Figure 5c, tenant-B first increased her allocation in link-2 to $\frac{1}{2}$ from $\frac{1}{9}$ and then work conservation increased her allocation in link-1 to $\frac{3}{4}$ from the initial $\frac{1}{2}$.

Algorithm 1 High Utilization with Guarantees (HUG)

```
Input: $\{\overrightarrow{d_k}\}$: reported correlation vector of tenant-$k$, $\forall k$
Output: $\{\overrightarrow{a_k}\}$: guaranteed resource allocated to tenant-$k$, $\forall k$

Stage 1: Calculate the optimal isolation guarantee ($\mathcal{M}^*$)
    and minimum allocations $\overrightarrow{a_k} = \mathcal{M}^* \overrightarrow{d_k}, \forall k$

Stage 2: Restrict maximum utilization for each of
    the $2P$ links, such that $c_k^i \le \mathcal{M}^*, \forall i, \forall k$
```

Consequently, we must eliminate a tenant's incentive to gain too much spare resources by lying; i.e., *a tenant should never be able to manipulate and increase her progress due to work conservation.*

Lemma 1 *Any allocation policy with the following two characteristics is not strategyproof:*

1. *it first uses DRF to ensure the optimal isolation guarantee and then assigns the spare, DRF-unallocated resources for work conservation;*
2. *there exists at least one tenant whose allocation (including spare) on some link is more than her progress under DRF based on her reported correlation vector.*

Corollary 2 (of Lemma 1) *Optimal isolation guarantee allocations cannot always be work-conserving even in the presence of elastic-demand applications.* □

3.2 The Optimal Algorithm: HUG

Given the tradeoff, our goal is to design an allocation algorithm that can achieve the highest utilization while keeping the optimal isolation guarantee and strategy-proofness. Formally, we want to design an algorithm to

$$\begin{aligned} \text{Maximize} \quad & \sum_{i \in [1,2P]} \sum_{k \in [1,M]} c_k^i \\ \text{subject to} \quad & \min_{k \in [1,M]} \mathcal{M}_k = \mathcal{M}^*, \end{aligned} \qquad (1)$$

where c_k^i is the actual consumption[7] of tenant-k on link-i for allocation a_k^i, and $\mathcal{M}^*$ is the optimal isolation guarantee.

We observe that an optimal algorithm would have restricted tenant-A's progress in Figure 5b and tenant-B's progress in Figure 5c to $\frac{2}{3}$. Consequently, they would not have been incentivized to lie and the prisoner's dilemma could have been avoided. Algorithm 1 – referred to as *High Utilization with Guarantees* (HUG) – is such a two-stage allocation mechanism that guarantees maximum utilization while maximizing the isolation guarantees across tenants and is strategyproof.

In the first stage, HUG allocates resources to maximize isolation guarantees across tenants. To achieve this, we pose our problem as a $2P$-resource fair sharing problem and use DRF [33, 56] to calculate $\mathcal{M}^*$. By reserving

[7] Consumptions can differ from allocations when tenants are lying.

these allocations, HUG ensures isolation. Moreover, because DRF is strategy-proof, tenants are guaranteed to use these allocations (i.e., $c_k^i \geq a_k^i$).

While DRF maximizes the isolation guarantees (a.k.a. dominant shares), it results in low network utilization. In some cases, *DRF may even have utilization arbitrarily close to zero, and HUG can increase that to 100%* (**Lemma 6**).

To achieve this, the second stage of HUG maximizes utilization while still keeping the allocation strategyproof. In this stage, we calculate upper bounds to restrict how much of the spare capacity a tenant can use in each link, with the constraint that the largest share across all links cannot increase (**Lemma 1**). As a result, Algorithm 1 remains strategy-proofness across both stages. Because spare usage restrictions can be applied locally, HUG can be enforced in individual machines.

Illustrated in Figure 8, the bound is set at $\frac{2}{3}$ for both tenants, and tenant-B can use its elastic demand on link-2's spare resource, while tenant-A cannot as she has reached its bound on link-2.

3.3 HUG Properties

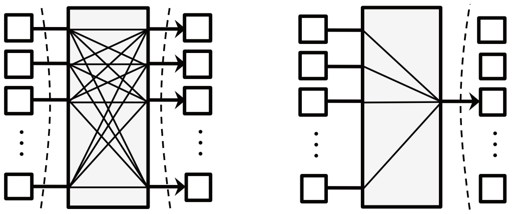
We list the main properties of HUG in the following.

1. In non-cooperative cloud environments, HUG is strategyproof (**Theorem 3**), maximizes isolation guarantees (**Corollary 4**), and ensures the highest utilization possible for an optimal isolation guarantee allocation (**Theorem 5**). In particular, **Lemma 6** shows that under some cases, DRF may have utilization arbitrarily close to 0, and HUG improves it to 100%. We defer the proofs of properties in the Section to Appendix A.
2. In cooperative environments like private datacenters, HUG maximizes isolation guarantees and is work-conserving. Work conservation is achievable because strategy-proofness is a non-requirement in this case.
3. Because HUG provides optimal isolation guarantee, it provides min-cut proportionality (§ 3.3.1) in both non-cooperative and cooperative environments.

Regardless of resource types, the identified tradeoff exists in general multi-resource allocation problems and HUG can directly be applied.

3.3.1 Min-Cut Proportionality

Prior work promoted the notion of *proportionality* [58], where tenants would expect to receive total allocations proportional to their number of VMs *regardless* of communication patterns. Meaning, two tenants, each with N VMs, should receive equal bandwidth even if tenant-X has an all-to-all communication pattern (i.e., $\overrightarrow{d_X} = \langle 1, 1, \ldots, 1 \rangle$) and tenant-Y has an N-to-1 pattern (i.e., exactly one 1 in $\overrightarrow{d_Y}$ and the rest are zeros). Figure 9 shows an example. Clearly, tenant-Y will be bottlenecked at her only receiver; trying to equalize them will only result in low utilization. As expected, FairCloud proved that such proportionality is not achievable as it decreases both isolation guarantee and utilization [58]. None of the existing algorithms provide proportionality.



(a) Tenant-X (N-to-N) **(b)** Tenant-Y (N-to-1)

Figure 9: Communication patterns of tenant-X and tenant-Y with (a) two minimum cuts of size P, where P is the number of fabric ports, and (b) one minimum cut of size 1. The size of the minimum cut of a communication pattern determines its effective bandwidth even if it were placed alone.

Instead, we consider a relaxed notion of proportionality, called *min-cut proportionality*, that depends on communication patterns and ties proportionality with a tenant's progress. Specifically, each tenant receives minimum bandwidth proportional to the size of the *minimum cut* [31] of their communication patterns. Meaning, in the earlier example, tenant-X would receive P times more total bandwidth than tenant-Y, but they would have the optimal isolation guarantee ($\mathcal{M}_X = \mathcal{M}_Y = \frac{1}{2}$).

Min-cut proportionality and optimal isolation guarantee can coexist, but they both have tradeoffs with work conservation.

4 Design Details

This section discusses how a cloud operator can implement, enforce, and expose HUG to the tenants (§4.1), how to exploit placement to further improve HUG's performance (§4.2), and how HUG can handle weighted, heterogeneous scenarios (§4.3).

4.1 Architectural Overview

HUG can easily be implemented atop existing monitoring infrastructure of cloud operators (e.g., Amazon CloudWatch [1]). Tenants would periodically update their correlation vectors through a public API, and the operator would compute new allocations and update enforcing agents within milliseconds.

HUG API The tenant-facing API simply transfers a tenant's correlation vector ($\overrightarrow{d_k}$) to the operator. $\overrightarrow{d_k} = \langle 1, 1, \ldots, 1 \rangle$ is used as the default correlation vector. By design, HUG incentivizes tenants to report and maintain accurate correlation vectors. This is because the more accurate it is – instead of the default $\overrightarrow{d_k} = \langle 1, 1, \ldots, 1 \rangle$ –

the higher are her progress and performance.

Many applications already know their long-term profiles (e.g., multi-tier online services [19, 52]) and others can calculate on the fly (e.g., bulk communication data-intensive applications [22, 23]). Moreover, existing techniques in traffic engineering can provide good accuracy in estimating and predicting demand matrices for coarse time granularities [17, 18, 20, 47, 48].

Centralized Computation For any update, the operator must run Algorithm 1. Although Stage-1 requires solving a linear program to determine the optimal isolation guarantee (i.e., the DRF allocation) [33], it can also be rewritten as a closed-form equation [56] when tenants can scale up and down following their normalized correlation vectors. The progress of all tenants after Stage-1 of Algorithm 1 – the optimal isolation guarantee – is:

$$\mathcal{M}^* = \frac{1}{\max\limits_{1 \le i \le 2P} \sum\limits_{k=1}^{M} d_k^i} \tag{2}$$

Equation (2) is computationally inexpensive. For our 100-machine cluster, calculating $\mathcal{M}^*$ takes about 5 microseconds. Communicating the decision to all 100 machines takes just 8 milliseconds and to $100,000$ (emulated) machines takes less than 1 second (§5.1.2).

The guaranteed minimum allocations of tenant-k can then be calculated as $a_k^i = \mathcal{M}^* d_k^i$ for all $1 \le i \le 2P$.

Local Enforcement Enforcement in Stage-2 of Algorithm 1 is simple as well. After reserving the minimum uplink and downlink allocations for each tenant, each machine needs to ensure that no tenant can consume more than $\mathcal{M}^*$ fraction of the machine's up or down link capacities (C^i) to the network; i.e., $a_k^i \le c_k^i \le \mathcal{M}^*$. The spare is allocated among tenants using local max-min fairness *subject to* tenant-specific upper-bounds.

Because we only care about inter-tenant behavior – not how a tenant performs internal sharing – stock Linux `tc` is sufficient (§5). A tenant has the flexibility to choose from traditional per-flow fairness, shortest-first flow scheduling [12, 41], or explicit rate-based flow control [29].

4.2 VM Placement and Re-Placement/Migration

While $\mathcal{M}^*$ is optimal for a *given* placement, it can be improved by changing the placement of tenant VMs based on their correlation vectors. One must perform load balancing across all machines to minimize the denominator of Equation (2). Cloud operators can employ optimization frameworks like [19] to perform initial VM placement and periodic migrations with an additional load balancing constraint. However, VM placement is a notoriously difficult problem because of often-incompatible constraints like fault-tolerance and collocation [19], and we consider its detailed study an important future work. It is worth noting that with *any* VM placement, HUG provides the highest attainable utilization without sacrificing optimal isolation guarantee and strategy-proofness.

4.3 Additional Constraints

Weighted Tenants Giving preferential treatment to tenants is simple. Just using $w_k\overrightarrow{d_k}$ instead of $\overrightarrow{d_k}$ in Equation (2) would account for tenant weights (w_k for tenant-k) in calculating $\mathcal{M}^*$.

Heterogeneous Capacities Because allocations are calculated independently in each machine based on $\mathcal{M}^*$ and local capacities (C^i), HUG supports heterogeneous link capacities.

Bounded Demands So far we have considered only elastic tenants. If tenant-k has bounded demands, i.e., $d_k^i < 1$ for all $i \in [1, 2P]$, calculating a common $\mathcal{M}^*$ and corresponding $\overrightarrow{a_k}$ in *one round* using Equation (2) will be inefficient. This is because tenant-k might require less than the calculated allocation, and being bounded, she cannot elastically scale up to use it. Instead, we must use the *multi-round* DRF algorithm [56, Algorithm 1] in Stage-1 of HUG; Stage-2 will remain the same. Note that this is similar to max-min fairness in a single link when a flow has a smaller demand than its $\frac{1}{n}$-th share.

5 Evaluation

We evaluated HUG using trace-driven simulations and EC2 deployments. Our results show the following:

- HUG isolates multiple tenants across the entire network, and it can scale up to $100,000$ machines with less than one second overhead (§5.1).
- HUG ensures the optimal isolation guarantee – almost 7000× more than per-flow fairness and about 7.4× more than PS-P in production traces – while providing 1.4× higher utilization than DRF (§5.2).
- HUG outperforms per-flow fairness (PS-P) by 17.35× (1.48×) in terms of the 95th percentile slowdown and by 1.49× (1.14×) in minimizing the average shuffle completion time (§5.3).
- HUG outperforms Varys [23] in terms of the maximum shuffle completion time by 1.77×, even though Varys is 1.45× better in minimizing the average shuffle completion time and 1.33× better in terms of the 95th percentile slowdown (§5.3).

We present our results in three parts. First, we microbenchmark HUG on 100-machine EC2 clusters to evaluate HUG's guarantees and overheads (§5.1). Second, we leverage traces collected from a 3200-machine Facebook cluster by Popa et al. [58] to compare HUG's *instantaneous* allocation characteristics with that of per-

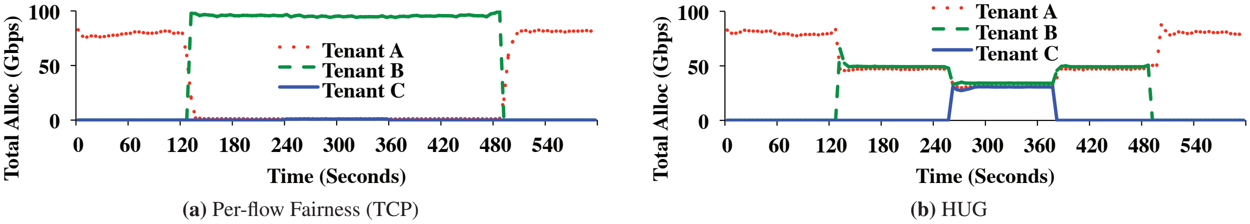


Figure 10: [EC2] Bandwidth consumptions of three tenants arriving over time in a 100-machine EC2 cluster. Each tenant has 100 VMs, but each uses a different communication pattern (§5.1.1). We observe that (a) using TCP, tenant-B dominates the network by creating more flows; (b) HUG isolates tenants A and C from tenant B.

flow fairness, PS-P [58], and DRF [33] (§5.2). Finally, we evaluate HUG's *long-term* impact on application performance using a 3000-machine Facebook cluster trace used by Chowdhury et al. [23] and compare against per-flow fairness, PS-P, DRF, as well as Varys, which focuses only on improving performance (§5.3).

5.1 Testbed Experiments

Methodology We performed our experiments on 100 `m2.4xlarge` Amazon EC2 [2] instances running on Linux kernel 3.4.37 and used the default `htb` and `tc` implementations. While there exist proposals for more accurate `qdisc` implementations [45, 57], the default `htb` worked sufficiently well for our purposes. Each of the machines had 1 Gbps NICs, and we could use close to full 100 Gbps bandwidth simultaneously.

5.1.1 Network-Wide Isolation

We consider a cluster with 100 EC2 machines, divided between three tenants A, B, and C that arrive over time. Each tenant has 100 VMs; i.e., VMs A_i, B_i, and C_i are collocated on the i-th physical machine. However, they have different communication patterns: tenants A and C have pairwise one-to-one communication patterns (100 VM-VM flows each), whereas tenant-B follows an all-to-all pattern using $10,000$ flows. Specifically, A_i communicates with $A_{(i+50)\%100}$, C_j communicates with $C_{(j+25)\%100}$, and any B_k communicates with all B_l, where $i, j, k, l \in \{1, ..., 100\}$. Each tenant demands the entire capacity at each machine; hence, the entire capacity of the cluster should be equally divided among the active tenants to maximize isolation guarantees.

Figure 10a shows that as soon as tenant-B arrives, she takes up the entire capacity in the absence of isolation guarantee. Tenant-C receives only marginal share as she arrives after tenant-B and leaves before her. Note that tenant-A (when alone) uses only about 80% of the available capacity; this is simply because just one TCP flow per VM-VM pair often cannot saturate the link.

Figure 10b presents the allocation using HUG. As tenants arrive and depart, allocations are dynamically calculated, propagated, and enforced in each machine of the cluster. As before, tenants A and C use marginally less than their allocations because of creating only one flow between each VM-VM pair.

5.1.2 Scalability

The key challenge in scaling HUG is its centralized resource allocator, which must recalculate tenant shares and redistribute them across the entire cluster whenever any tenant changes her correlation vector.

We found that the time to calculate new allocations using HUG is less than 5 microseconds in our 100 machine cluster. Furthermore, a recomputation due to a tenant's arrival, departure, or change of correlation vector would take about 8.6 milliseconds on average for a $100,000$-machine datacenter.

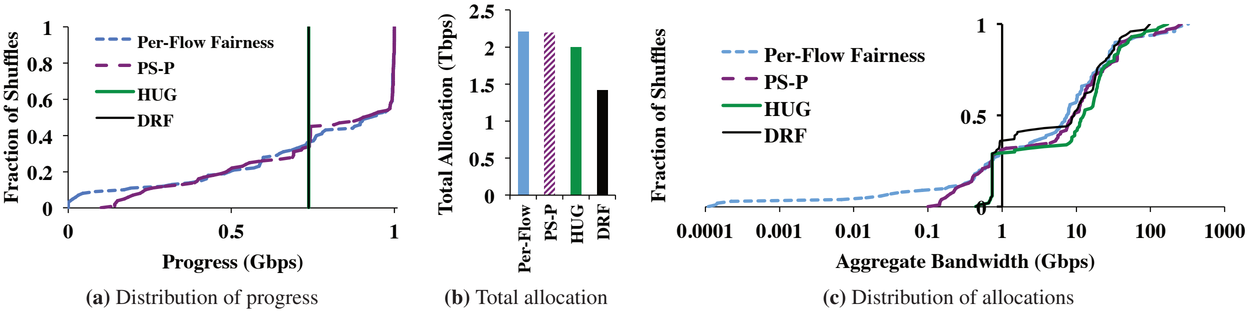
Communicating a new allocation takes less than 10 milliseconds to 100 machines and around 1 second for $100,000$ *emulated* machines (i.e., sending the same message 1000 times to each of the 100 machines).

5.2 Instantaneous Fairness

While Section 5.1 evaluated HUG in controlled, synthetic scenarios, this section focuses on HUG's instantaneous allocation characteristics in the context of a large-scale cluster.

Methodology We use a one-hour snapshot with 100 concurrent jobs from a production MapReduce trace, which was extracted from a 3200-machine Facebook cluster by Popa et al. [58, Section 5.3]. Machines are connected to the network using 1 Gbps NICs. In the trace, a job with M mappers and R reducers – hence, the corresponding $M \times R$ shuffle – is described as a matrix with the amount of data to transfer between each M-R pair. We calculated the correlation vectors of individual shuffles from their communication matrices ourselves using the optimal rate allocation algorithm for a single shuffle [22, 23], ensuring *all* the flows of each shuffle to finish simultaneously.

Given the workload, we calculate progress of each job/shuffle using different allocation mechanisms and



(a) Distribution of progress **(b)** Total allocation **(c)** Distribution of allocations

Figure 11: [Simulation] HUG ensures higher isolation guarantee than high-utilization schemes like per-flow fairness and PS-P, and provides higher utilization than multi-resource fairness schemes like DRF.

cross-examine characteristics like isolation guarantee, utilization, and proportionality.

5.2.1 Impact on Progress

Figure 11a presents the distribution of progress of each shuffle. Recall that the progress of a shuffle – we consider each shuffle an individual tenant in this section – is the amount of bandwidth it is receiving in its bottleneck up or downlink (i.e., progress can be at most 1 Gbps). Both HUG and DRF (overlapping vertical lines in Figure 11a) ensure the same progress (0.74 Gbps) for all shuffles. Note that despite same progress, shuffles will finish at different times based on how much data each one has to send (§5.3). Per-flow fairness and PS-P provide very wide ranges: 112 Kbps to 1 Gbps for the former and 0.1 Gbps to 1 Gbps for the latter. Shuffles with many flows crowd out the ones with fewer flows under per-flow fairness, and PS-P suffers by ignoring correlation vectors and through indiscriminate work conservation.

5.2.2 Impact on Utilization

By favoring heavy tenants, per-flow fairness and PS-P do succeed in their goals of increasing network utilization (Figure 11b). Given the communication patterns of the workload, the former utilizes 69% of 3.2 Tbps total capacity across all machines and the latter utilizes 68.6%. In contrast, DRF utilizes only 45%. HUG provides a common ground by extending utilization to 62.4% without breaking strategy-proofness and providing optimal isolation guarantee.

Figure 11c breaks down total allocations of each shuffle and demonstrates two high-level points:

1. HUG ensures overall higher utilization (1.4× on average) than DRF by ensuring equal progress for smaller shuffles and by using up additional bandwidth for larger shuffles. It does so while ensuring the same optimal isolation guarantee as DRF.
2. Per-flow fairness crosses HUG at the 90-th percentile; i.e., the top 10% shuffles receive more bandwidth than they do under HUG, while the other 90% receive less than they do using HUG. PS-P crosses over at the 76-th percentile.

Bin	1 (SN)	2 (LN)	3 (SW)	4 (LW)
% of Shuffles	52%	16%	15%	17%

Table 2: Shuffles binned by their lengths (**S**hort and **L**ong) and widths (**N**arrow and **W**ide).

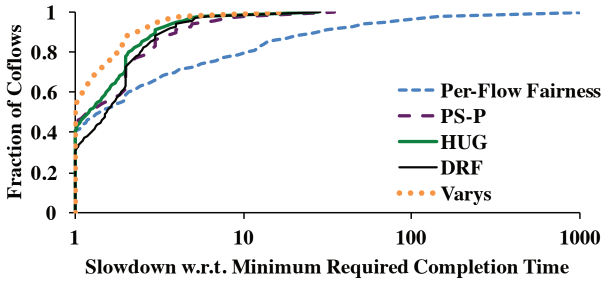
5.2.3 Impact on Proportionality

A collateral benefit of HUG is that tenants receive allocations proportional to their bottleneck demands. Consequently, despite the same progress across all shuffles (Figure 11a), their total allocations vary (Figure 11c) based on the size of minimum cuts in their communication patterns.

5.3 Long-Term Characteristics

We have shown in Section 5.2 that HUG provides optimal isolation guarantee in the *instantaneous* case. However, similar to all instantaneous solutions [33, 43, 58], HUG does not provide any *long-term* isolation or fairness guarantees. Consequently, in this section, we evaluate HUG's long-term impact on performance using a production trace through simulations.

Methodology For these simulations, we use a MapReduce/Hive trace from a 3000-machine production Facebook cluster. The trace includes the arrival times, communication matrices, and placements of tasks of over 10,000 shuffle during one day. Shuffles in this trace have diverse length (i.e., size of the longest flow) and width (i.e., the number of flows) characteristics and roughly follow the same distribution of the original trace (Table 2). We consider a shuffle to be *short* if its longest flow is less than 5 MB and *narrow* if it has at most 50 flows; we use the same categorization. We calculated the correlation vector of each shuffle as we did before (§ 5.2).



(a) CDFs

	Min	95th	AVG	STDEV
Per-Flow Fairness	1	69.4	15.52	65.54
PS-P	1	5.9	2.22	2.97
HUG	1	4.0	1.86	2.25
DRF	1	4.4	2.11	2.66
Varys	1	3.0	1.43	0.99

(b) Summaries

Figure 12: [Simulation] Slowdowns of shuffles using different mechanisms w.r.t. the minimum completion times of each shuffle. The X-axis of (a) is in log scale.

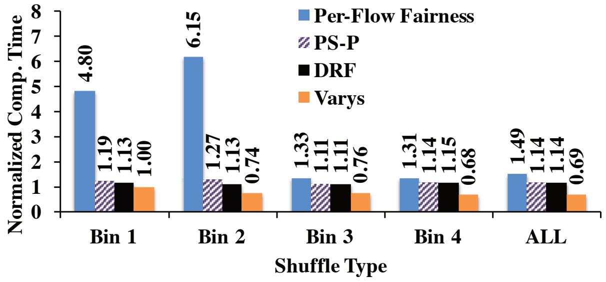


Figure 13: [Simulation] Average shuffle completion times normalized by HUG's average completion time. 95-th percentile plots are similar.

Metrics We consider two metrics: 95*th percentile slowdown* and *average shuffle completion time* to respectively measure long-term progress and performance characteristics.

We define the slowdown of a shuffle as its completion time due to a scheme normalized by its minimum completion time if it were running alone; i.e.,

$$\text{Slowdown} = \frac{\text{Compared Duration}}{\text{Minimum Duration}}$$

The minimum value of slowdown is one.

We measure performance as the shuffle completion time of a scheme normalized by that using HUG; i.e.,

$$\text{Normalized Comp. Time} = \frac{\text{Compared Duration}}{\text{HUG's Duration}}$$

If the normalized completion time of a scheme is greater (smaller) than one, HUG is faster (slower).

5.3.1 Improvements Over Per-Flow Fairness

HUG improves over per-flow fairness both in terms of slowdown and performance. The 95th percentile slowdown using HUG is 17.35× better than that of per-flow fairness (Table 12b). Overall, HUG provides better slowdown across the board (Figure 12a) – 61% shuffles are better off using HUG and the rest remain the same.

HUG improves the average completion time of shuffles by 1.49× (Figure 13). The biggest wins comes from bin-1 (4.8×) and bin-2 (6.15×) that include the so-called narrow shuffles with less than 50 flows. This reinforces the fact that HUG isolates tenants with fewer flows from those with many flows. Overall, HUG performs well across all bins.

5.3.2 Improvements Over PS-P

HUG improves over PS-P in terms of the 95th percentile slowdown by 1.48×, and 45% shuffles are better off using HUG. HUG also providers better average shuffle completion times than PS-P for an overall improvement of 1.14×. Large improvements again come in bin-1 (1.19×) and bin-2 (1.27×) because PS-P also favors tenants with more flows.

Note that instantaneous high utilization of per-flow fairness and PS-P (§5.2) does not help in the long run due to lower isolation guarantee.

5.3.3 Improvements Over DRF

While HUG and DRF has the same worst-case slowdown, 70% shuffles are better off using HUG. HUG also provides better average shuffle completion times than DRF for an overall improvement of 1.14×.

5.3.4 Comparison to Varys

Varys outperforms HUG by 1.33× in terms of the 95th percentile slowdown and by 1.45× in terms of average shuffle completion time. However, because Varys attempts to improve the average completion time by prioritization, it risks in terms of the maximum completion time. More precisely, HUG outperforms Varys by 1.77× in terms of the maximum shuffle completion time (not shown).

6 Discussion

Payment Model Similar to many existing proposals [32, 33, 45, 46, 58, 59, 61, 62], we assume that tenants pay per-hour flat rates for individual VMs, but there is no pricing model associated with their network usage. This is also the prevalent model of resource pricing in cloud computing [2, 5, 8]. Exploring *whether* and *how* a network pricing model would change our solution and *what* that model would look like requires further attention.

Determining Correlation Vectors Unlike long-term correlation vectors, e.g., over the course of an hour or for an entire shuffle, accurately capturing short-term changes can be difficult. How fast tenants should update their vectors and whether that is faster than centralized HUG can react to requires additional analysis.

Decentralized HUG HUG's centralized design makes it easier to analyze its properties and simplifies its implementation. We believe that designing a decentralized version of HUG is an important future work, which will be especially relevant for sharing wide-area networks in the context of geo-distributed analytics [60, 66].

7 Related Work

Single-Resource Fairness Max-min fairness was first proposed by Jaffe [43] to ensure at least $\frac{1}{n}$-th of a link's capacity to each flow. Thereafter, many mechanisms have been proposed to achieve it, including weighted fair queueing (WFQ) [25, 55] and those similar to or extending WFQ [16, 34, 35, 63, 64]. We generalize max-min fairness to parallel communication observed in scale-out applications, showing that unlike in the single-link scenario, optimal isolation guarantee, strategy-proofness, and work conservation cannot coexist.

Multi-Resource Fairness Dominant Resource Fairness (DRF) [33] maximizes the dominant share of each user in a strategyproof manner. Solutions that have attempted to improve the system-level efficiency of multi-resource allocation – both before [54, 65] and after [27, 38, 56] DRF – sacrifice strategy-proofness. We have proven that work-conserving allocation without strategy-proofness can hurt utilization instead of improving it.

Dominant Resource Fair Queueing (DRFQ) [32] approximates DRF over time in *individual* middleboxes. In contrast, HUG generalizes DRF to environments with elastic demands to increase utilization across the *entire* network and focuses only on instantaneous fairness.

Joe-Wong et al. [46] have presented a unifying framework to capture fairness-efficiency tradeoffs in multi-resource environments. They assume a *cooperative* environment, where tenants never lie. HUG falls under their FDS family of mechanisms. In non-cooperative environments, however, we have shown that the interplay between work conservation and strategy-proofness is critical, and our work complements the framework of [46].

Network-Wide / Tenant-Level Fairness Proposals for sharing cloud networks range from static allocation [13, 14, 44] and VM-level guarantees [61, 62] to variations of network-wide sharing mechanisms [45, 51, 58, 59, 67]. We refer the reader to the survey by Mogul and Popa [53] for an overview. FairCloud [58] stands out by systematically discussing the tradeoffs and addresses several limitations of other approaches. Our work generalizes FairCloud [58] and many proposals similar to FairCloud's PS-P policy [45, 59, 61]. When all tenants have elastic demands, i.e., all correlation vectors have all elements as 1, we give the same allocation; for all other cases, we provide higher isolation guarantee and utilization.

Efficient Schedulers Researchers have also focused on efficient scheduling and/or packing of datacenter resources to minimize job and communication completion times [12, 21–23, 26, 36, 41]. Our work is orthogonal and complementary to these work focusing on application-level efficiency within each tenant. We guarantee isolation across tenants, so that each tenant can internally perform whatever efficiency or fairness optimizations among her own applications.

8 Conclusion

In this paper, we have proved that there is a strong tradeoff between optimal isolation guarantees and high utilization in non-cooperative public clouds. We have also proved that work conservation can decrease utilization instead of improving it, because no network sharing algorithm remains strategyproof in its presence.

To this end, we have proposed HUG to restrict bandwidth utilization of each tenant to ensure highest utilization with optimal isolation guarantee across multiple tenants in non-cooperative environments. In cooperative environments, where strategy-proofness might be a non-requirement, HUG simultaneously ensures both work conservation and the optimal isolation guarantee.

HUG generalizes single-resource max-min fairness to multi-resource environments where a tenant's demand on different resources are correlated and elastic. In particular, it provides optimal isolation guarantee, which is significantly higher than that provided by existing multi-tenant network sharing algorithms. HUG also complements DRF with provably highest utilization without sacrificing other useful properties of DRF. Regardless of resource types, the identified tradeoff exists in general multi-resource allocation problems, and all those scenarios can take advantage of HUG.

Acknowledgments

We thank our shepherd Mohammad Alizadeh and the anonymous reviewers of SIGCOMM'15 and NSDI'16 for useful feedback. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, NSF Award CNS-1464388, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, EMC2, Ericsson, Facebook, Fujitsu, Guavus, HP, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata, and VMware.

References

[1] Amazon CloudWatch. http://aws.amazon.com/cloudwatch.

[2] Amazon EC2. http://aws.amazon.com/ec2.

[3] Apache Hadoop. http://hadoop.apache.org.

[4] AWS Innovation at Scale. http://goo.gl/Py2Ueo.

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

[6] Google Container Engine. http://kubernetes.io.

[7] A look inside Google's data center networks. http://goo.gl/u0vZCY.

[8] Microsoft Azure. http://azure.microsoft.com.

[9] Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net.

[10] Trident: Stateful Stream Processing on Storm. http://goo.gl/cKsvbj.

[11] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*, 2014.

[12] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. Mckeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM*, 2013.

[13] S. Angel, H. Ballani, T. Karagiannis, G. OShea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *OSDI*, 2014.

[14] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[15] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. OShea. Chatty tenants and the cloud network sharing problem. In *NSDI*, 2013.

[16] J. Bennett and H. Zhang. WF^{2}Q: Worst-case fair weighted fair queueing. In *INFOCOM*, 1996.

[17] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[18] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *CoNEXT*, 2011.

[19] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[20] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM*, 2013.

[21] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[22] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[23] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.

[24] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[25] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, 1989.

[26] F. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *SIGCOMM*, 2014.

[27] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial. No justified complaints: On fair sharing of multiple resources. In *ITCS*, 2012.

[28] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. In *SIGCOMM*, 1999.

[29] N. Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. PhD thesis, Stanford University, 2007.

[30] M. M. Flood. Some experimental games. *Management Science*, 5(1):5–26, 1958.

[31] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[32] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. *SIGCOMM*, 2012.

[33] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[34] S. J. Golestani. Network delay analysis of a class of fair queueing algorithms. *IEEE JSAC*, 13(6):1057–1070, 1995.

[35] P. Goyal, H. M. Vin, and H. Chen. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *SIGCOMM*, 1996.

[36] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[37] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *CoNEXT*, 2010.

[38] A. Gutman and N. Nisan. Fair allocation without trade. In *AAMAS*, 2012.

[39] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. In *SIGCOMM*, 2010.

[40] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[41] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM*, 2012.

[42] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[43] J. M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[44] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message completion time in the cloud. In *SIGCOMM*, 2015.

[45] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical network performance isolation at the edge. In *NSDI*, 2013.

[46] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multiresource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM*, 2012.

[47] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *SIGCOMM*, 2005.

[48] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: Measurements and analysis. In *IMC*, 2009.

[49] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" abstraction in Software-Defined Networks. In *CoNEXT*, 2013.

[50] K. LaCurts, J. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *HotCloud*, 2014.

[51] V. T. Lam, S. Radhakrishnan, A. Vahdat, G. Varghese, and R. Pan. NetShare and stochastic NetShare: Predictable bandwidth allocation for data centers. *SIGCOMM CCR*, 42(3):5–11, 2012.

[52] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *SIGCOMM*, 2014.

[53] J. C. Mogul and L. Popa. What we talk about when we talk about cloud network performance. *SIGCOMM CCR*, 42(5):44–48, 2012.

[54] J. F. Nash Jr. The bargaining problem. *Econometrica: Journal of the Econometric Society*, pages 155–162, 1950.

[55] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM ToN*, 1(3):344–357, 1993.

[56] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond dominant resource fairness: extensions, limitations, and indivisibilities. In *EC*, 2012.

[57] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *SIGCOMM*, 2014.

[58] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

[59] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing. In *SIGCOMM*, 2013.

[60] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, V. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, 2015.

[61] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *USENIX WIOV*, 2011.

[62] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Sharing the data center network. In *NSDI*, 2011.

[63] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM*, 1995.

[64] I. Stoica, H. Zhang, and T. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. In *SIGCOMM*, 1997.

[65] H. R. Varian. Equity, envy, and efficiency. *Journal of economic theory*, 9(1):63–91, 1974.

[66] A. Vulimiri, C. Curino, B. Godfrey, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[67] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. In *SIGCOMM*, 2012.

[68] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[69] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

A Proofs from Section 3

Proof Sketch (of Lemma 1) Consider tenant-A from the example in Figure 5. Assume that instead of reporting her true correlation vector $\overrightarrow{d_A} = \langle \frac{1}{2}, 1 \rangle$, she reports $\overrightarrow{d'_A} = \langle \frac{1}{2} + \epsilon, 1 \rangle$, where $\epsilon > 0$. As a result, her allocation will change to $\overrightarrow{a'_A} = \langle \frac{1/2+\epsilon}{3/2+\epsilon}, \frac{1}{3/2+\epsilon} \rangle$. Her allocation in link-1 $\left(\frac{1/2+\epsilon}{3/2+\epsilon} \right)$ is already larger than before $\left(\frac{1}{3} \right)$. If the work conservation policy allocates the spare resource in link-2 by δ (δ may be small but a positive value), her progress will change to $\mathcal{M}'_A = \min \left(\frac{{a'}_A^1}{d_A^1}, \frac{{a'}_A^2}{d_A^2} \right) = \min \left(\frac{1+2\epsilon}{3/2+\epsilon}, \frac{1}{3/2+\epsilon} + \delta \right)$. As long as $\epsilon < \frac{3/2\delta}{2/3-\delta}$ (if $\delta \geq \frac{2}{3}$, we have no constraint on ϵ), her progress will be better than when she was telling the truth, which makes the policy not strategyproof. The operator cannot prevent this because she knows neither a tenant's true correlation vector nor ϵ, the extent of the tenant's lie. □

Theorem 3 *Algorithm 1 is strategyproof.*

Proof Sketch (of Theorem 3) Because DRF is strategyproof, the first stage of Algorithm 1 is strategyproof as well. We show that adding the second stage does not violate strategy-proofness of the combination.

Assume that link-b is a system bottleneck – the link DRF saturated to maximize isolation guarantee in the first stage. Meaning, $b = \arg\max_i \sum_{k=1}^{M} d_k^i$. We use $D^b = \sum_{k=1}^{M} d_k^b$ to denote the total demand in link-b ($D^b \geq 1$), and $\mathcal{M}_k^b = 1/D^b$ for corresponding progress for all tenant-k ($k \in \{1, ..., M\}$) when link-b is the system bottleneck. In Figure 5, $b = 1$. The following arguments hold even for multiple bottlenecks.

Any tenant-k can attempt to increase her progress ($\mathcal{M}_k$) only by lying about her correlation vector ($\overrightarrow{d_k}$). Formally, her action space consists of all possible correlation vectors. It includes increasing and/or decreasing demands of individual resources to report a different vector, $\overrightarrow{d'_k}$ and obtain a new progress, $\mathcal{M}'_k (> \mathcal{M}_k)$. Tenant-$k$ can attempt one of the two alternatives when reporting $\overrightarrow{d'_k}$: either keep link-b still the system bottleneck or change it. We show that Algorithm 1 is strategyproof in both cases; i.e., $\mathcal{M}'_k \leq \mathcal{M}_k$.

Case 1: link-b is still the system bottleneck.

Her progress cannot improve because

- if $d_k'^b \leq d_k^b$, her share on the system bottleneck will decrease in the first stage; so will her progress. There is no spare resource to allocate in link-b.

For example, if tenant-A changes $d_A'^1 = \frac{1}{4}$ instead of $d_A^1 = \frac{1}{2}$ in Figure 5, her allocation will decrease to $\frac{1}{5}$-th of link-1; hence, $\mathcal{M}'_A = \frac{2}{5}$ instead of $\mathcal{M}_A = \frac{2}{3}$.

- if $d_k'^b > d_k^b$, her share on the system bottleneck will increase. However, because $D'^b > D^b$ as $d_k'^b > d_k^b$, everyone's progress including her own will decrease in the first stage ($\mathcal{M}_k'^b \leq \mathcal{M}_k^b$). The second stage will ensure that her maximum consumption in any link-i $c_k'^i \leq \max_j \left\{ a_k'^j \right\}$. Therefore her progress will be smaller than that when she tells the truth ($\mathcal{M}'^b_k < \mathcal{M}^b_k$).

For example, if tenant-A changes $d'^1_A = 1$ instead of $d^1_A = \frac{1}{2}$ in Figure 5, her allocation will increase to $\frac{1}{2}$ of link-1. However, progress of both tenants will decrease: $\mathcal{M}_A = \mathcal{M}_B = \frac{1}{2}$. The second stage will restrict her usage in link-2 to $\frac{1}{2}$ as well; hence, $\mathcal{M}'_A = \frac{1}{2}$ instead of $\mathcal{M}_A = \frac{2}{3}$.

Case 2: link-b is no longer a system bottleneck; instead, link-b' ($\neq b$) is now one of the system bottlenecks.

We need to consider the following two sub-cases.

- If $D'^{b'} \leq D^b$, the progress in the first stage will increase; i.e., $\mathcal{M}'^{b'}_k \geq \mathcal{M}^b_k$. However, tenant-$k$'s allocation in link-$b$ will be no larger than if she had told the truth, making her progress no better. To see this, consider the allocations of all other tenants in link-b before and after she lies. Denote by c^b_{-k} and c'^b_{-k} the resource consumption of all other tenants in link-b when tenant-k was telling the truth and lying, respectively. We also have $c^b_{-k} = a^b_{-k}$ and $a^b_{-k} + a^b_k = 1$ because link-b was the bottleneck, and there was no spare resource to allocate for this link. When tenant-k lies, $a'^b_{-k} \geq a^b_{-k}$ because $\mathcal{M}'^{b'}_k \geq \mathcal{M}^b_k$. We also have $c'^b_{-k} \geq a'^b_{-k}$ and $c'^b_{-k} + c'^b_k \leq 1$. This implies $c'^b_k \leq 1 - c'^b_{-k} \leq 1 - a'^b_{-k} \leq 1 - a^b_{-k} = a^b_k = c^b_k$. Meaning, tenant-$k$'s progress is no larger than that when she was telling the truth.

- If $D'^{b'} > D^b$, everyone's progress including her own decreases in the first stage ($\mathcal{M}'^{b'}_k < \mathcal{M}^b_k$). Similar to the second scenario in Case 1, the second stage will restrict tenant-k to the lowered progress.

Regardless of tenant-k's approaches – keeping the same system bottleneck or not – her progress using Algorithm 1 will not increase. □

Corollary 4 (of Theorem 3) *Algorithm 1 maximizes isolation guarantee, i.e., the minimum progress across tenants.* □

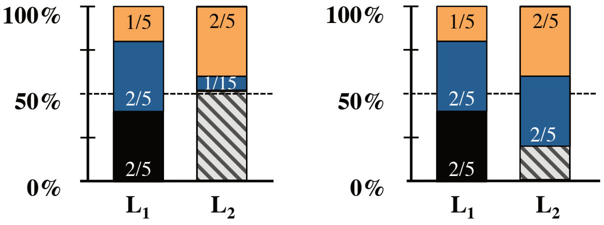
Theorem 5 *Algorithm 1 achieves the highest resource utilization among all strategyproof algorithms that provide optimal isolation guarantee among tenants.*

Proof Sketch (of Theorem 5) Follows from Lemma 1 and Theorem 3. □

Lemma 6 *Under some cases, DRF may have utilization arbitrarily close to 0, and HUG helps improve the utilization to 1.*

Proof Sketch (of Lemma 6) Construct the cases with K links and N tenants, and each tenant has demand 1 on link-1 and ϵ on other links.

DRF will allocate to each tenant $\frac{1}{N}$ on link-1 and $\frac{\epsilon}{N}$ on all other links, resulting in a total utilization of $\frac{1+(K-1)\epsilon}{K} \to 0$ when $K \to \infty, \epsilon \to 0$ for any N.



(a) DRF (b) HUG

Figure 14: Hard tradeoff between work conservation and strategy-proofness. Adding one more tenant (tenant-C in black) to Figure 5 with correlation vector $\langle 1, 0 \rangle$ makes simultaneously achieving work conservation and optimal isolation guarantee impossible, even when all three have elastic demands.

HUG will allocate to each tenant $\frac{1}{N}$ on every link and achieve 100% utilization. □

B Tradeoff Between Work Conservation and Strategy-proofness

We demonstrate the tradeoff between work conservation and strategy-proofness (thus isolation guarantee) by extending our running example from Section 2.

Consider another tenant (tenant-C) with correlation vector $\overrightarrow{d_C} = \langle 1, 0 \rangle$ in addition to the two tenants present earlier. The key distinction between tenant-C and either of the earlier two is that she does not demand any bandwidth on link-2. Given the three correlation vectors, we can use DRF to calculate the optimal isolation guarantee (Figure 14a), where tenant-k has $\mathcal{M}_k = \frac{2}{5}$, link-1 is completely utilized, and $\frac{7}{15}$-th of link-2 is proportionally divided between tenant-A and tenant-B.

This leaves us with two questions:

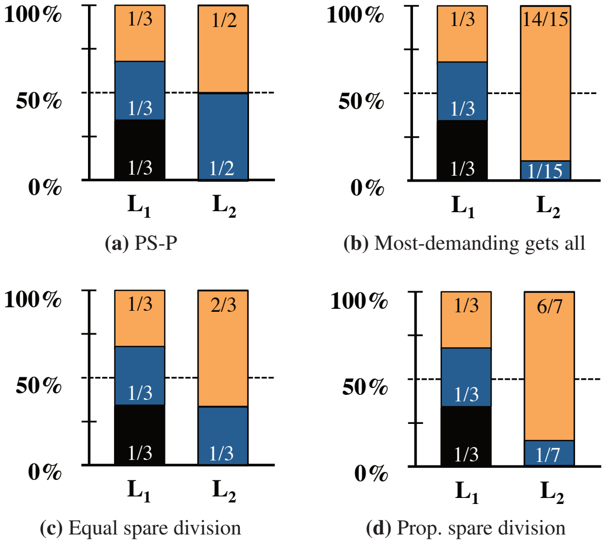
1. *How do we completely allocate the remaining $\frac{8}{15}$-th bandwidth of link-2?*
2. *Is it even possible without sacrificing optimal isolation guarantee and strategy-proofness?*

We show in the following that it is indeed not possible to allocate more than $\frac{4}{5}$-th of link-2 (Figure 14b) without sacrificing the optimal isolation guarantee.

Let us consider three primary categories of work-conserving spare allocation policies: *demand-agnostic*, *unfair*, and *locally fair*. All three will result in lower isolation guarantee, lower utilization, or both.

B.1 Demand-Agnostic Policies

Demand-agnostic policies equally divide the resource between the number of tenants independently in each link, irrespective of tenant demands, and provide isolation. Although strategyproof, this allocation (Figure 15a) has lower isolation guarantee ($\mathcal{M}_A = \frac{1}{2}$ and $\mathcal{M}_B = \mathcal{M}_C = \frac{1}{3}$, therefore isolation guarantee is $\frac{1}{3}$) than the op-



(a) PS-P

(b) Most-demanding gets all

(c) Equal spare division

(d) Prop. spare division

Figure 15: Allocations after applying different work-conserving policies to divide spare capacities in link-2 for the example in Figure 14.

timal isolation guarantee allocation shown in Figure 14a ($\mathcal{M}_A = \mathcal{M}_B = \mathcal{M}_C = \frac{2}{5}$, therefore isolation guarantee is $\frac{2}{5}$). PS-P [45, 58, 59] fall in this category.

Worse, when tenants do not have elastic-demand applications, demand-agnostic policies are not even work-conserving (similar to the example in §2.3.4).

Lemma 7 *When tenants do not have elastic demands, per-resource equal sharing is not work-conserving.*

Proof Sketch Only $\frac{11}{12}$-th of link-1 and $\frac{5}{9}$-th of link-2 will be consumed; i.e., none of the links will be saturated! □

To make it work-conserving, PS-P suggests dividing spare resources based on whoever wants it.

Lemma 8 *When tenants do not have elastic demands, PS-P is not work-conserving.*

Proof Sketch If tenant-B gives up her spare allocation in link-2, tenant-A can increase her progress to $\mathcal{M}_A = \frac{2}{3}$ and saturate link-1; however, tenant-B and tenant-C will remain at $\mathcal{M}_B = \mathcal{M}_C = \frac{1}{3}$. If tenant-$A$ gives up her spare allocation in link-1, tenant-B and tenant-C can increase their progress to $\mathcal{M}_B = \mathcal{M}_C = \frac{3}{8}$ and saturate link-1, but tenant-A will remain at $\mathcal{M}_A = \frac{1}{2}$. Because both tenant-$A$ and tenant-B have chances of increasing their progress, both will hold off to their allocations even with useless traffic – another instance of Prisoner's dilemma. □

B.2 Unfair Policies

Instead of demand-agnostic policies, one can also consider simpler, unfair policies; e.g., allocating all the resources to the tenant with the least or the most demand.

Lemma 9 *Allocating spare resource to the tenant with the least demand can result in zero spare allocation.*

Proof Sketch Although this strategy provides the optimal allocation for Figure 5, when at least one tenant in a link has zero demand, it can trivially result in no additional utilization; e.g., tenant-C in Figure 14. □

Lemma 10 *Allocating spare resource to the tenant with the least demand is not strategyproof.*

Proof Sketch Consider tenant-A lied and changed her correlation vector to $\overrightarrow{d'_A} = \langle 1, \frac{1}{10} \rangle$. The new optimal isolation guarantee allocation for unchanged tenant-B and tenant-C correlation vectors would be: $\overrightarrow{a_A} = \langle \frac{1}{3}, \frac{1}{30} \rangle$, $\overrightarrow{a_B} = \langle \frac{1}{3}, \frac{1}{18} \rangle$, and $\overrightarrow{a_C} = \langle \frac{1}{3}, 0 \rangle$. Now the spare resource in link-2 will be allocated to tenant-A because she asked for the least amount, and her final allocation would be $\overrightarrow{a'_A} = \langle \frac{1}{3}, \frac{17}{18} \rangle$. As a result, her progress improved from $\mathcal{M}_A = \frac{2}{5}$ to $\mathcal{M}'_A = \frac{2}{3}$, while the others' decreased to $\mathcal{M}_B = \mathcal{M}_C = \frac{1}{3}$. □

Corollary 11 (of Lemma 10) *In presence of work conservation, tenants can lie both by increasing and decreasing their demands, or a combination of both.* □

Lemma 12 *Allocating spare resource to the tenant with the highest demand is not strategyproof.*

Proof Sketch If tenant-A changes her correlation vector to $\overrightarrow{d'_A} = \langle 1, 1 \rangle$, the eventual allocation (Figure 15b) will again result in lower progress ($\mathcal{M}_B = \mathcal{M}_C = \frac{1}{3}$). Because tenant-$B$ is still receiving more than $\frac{1}{6}$-th of her allocation in link-1 in link-2, she does not need to lie. □

Corollary 13 (of Lemmas 10, 12) *Allocating spare resource randomly to tenants is not strategyproof.* □

B.3 Locally Fair Policies

Finally, one can also consider equally or proportionally dividing the spare resource on link-2 between tenant-A and tenant-B. Unfortunately, these strategies are not strategyproof either.

Lemma 14 *Allocating spare resource equally to tenants is not strategyproof.*

Proof Sketch If the remaining $\frac{8}{15}$-th of link-2 is equally divided, the share of tenant-A will increase to $\frac{2}{3}$-rd and incentivize her to lie. Again, the isolation guarantee will be smaller (Figure 15c). □

Lemma 15 *Allocating spare resource proportionally to tenants' demands is not strategyproof.*

Proof Sketch If one divides the spare in proportion to tenant demands, the allocation is different (Figure 15d) than equal division. However, tenant-A can again increase her progress at the expense of others. □

C Dual Objectives of Network Sharing

The two conflicting requirements of the network sharing problem can be defined as follows.

1. Utilization: $\sum_{i\in[1,2P]}\sum_{k\in[1,M]} c_k^i$
2. Isolation guarantee: $\min_{k\in[1,M]} \mathcal{M}_k$

Given the tradeoff between the two, one can consider one of the two possible optimizations:[8]

O1 Ensure highest utilization, *then* maximize the isolation guarantee with best effort;

O2 Ensure optimal isolation guarantee, *then* maximize utilization with best effort.

O1: Utilization-First In this case, the optimization attempts to maximize the isolation guarantee across all tenants while keeping the highest network utilization.

$$\begin{aligned} &\text{Maximize} && \min_{k\in[1,M]} \mathcal{M}_k \\ &\text{s.t.} && \sum_{i\in[1,2P]}\sum_{k\in[1,M]} c_k^i = U^*, \end{aligned} \tag{3}$$

where $U^* = \max \sum_{i\in[1,2P]}\sum_{k\in[1,M]} c_k^i$ is the highest utilization possible. Although this ensures maximum network utilization, isolation guarantee to individual tenants can be arbitrarily low. This formulation can still be useful in private datacenters [36].

To ensure some isolation guarantee, existing cloud network sharing approaches [14, 45, 51, 58, 59, 61, 62, 67] use a similar formulation:

$$\begin{aligned} &\text{Maximize} && \sum_{1\le i\le 2P}\sum_{k\in[1,M]} c_k^i \\ &\text{subject to} && \mathcal{M}_k \ge \frac{1}{M},\ k\in[1,M] \end{aligned} \tag{4}$$

The objective here is to maximize utilization while ensuring at least $\frac{1}{M}$-th of each link to tenant-k. However, this approach has two primary drawbacks (§ 2.3):

1. suboptimal isolation guarantee, and
2. lower utilization.

O2: Isolation-Guarantee-First Instead, in this paper, we have formulated the network sharing problem as follows:

$$\begin{aligned} &\text{Maximize} && \sum_{i\in[1,2P]}\sum_{k\in[1,M]} c_k^i \\ &\text{subject to} && \min_{k\in[1,M]} \mathcal{M}_k = \mathcal{M}_k^* \\ & && c_k^i \ge a_k^i,\ i\in[1,2P], k\in[1,M] \end{aligned} \tag{5}$$

Here, we maximize resource consumption while keeping the optimal isolation guarantee across all tenants, denoted by $\mathcal{M}_k^*$. Meanwhile, the constraint on consumption being at least guaranteed minimum allocation ensures strategy-proofness; thus, guaranteeing that guaranteed allocated resources will be utilized.

Because c_k^i values have no upper bounds except for physical capacity constraints, optimization **O2** may result in suboptimal isolation guarantee in non-cooperative environments (§2.3.3). HUG introduces the following additional constraint to avoid this issue only in non-cooperative environments:

$$c_k^i \le \mathcal{M}^*,\ i\in[1,2P], k\in[1,M]$$

This constraint is not necessary when strategy-proofness is a non-requirement – e.g., in private datacenters.

[8] Maximizing a combination of these two is also an interesting future direction.

Consensus in a Box: Inexpensive Coordination in Hardware

Zsolt István, David Sidler, Gustavo Alonso
Systems Group, Dept. of Computer Science, ETH Zürich

Marko Vukolić[*]
IBM Research - Zürich

Abstract

Consensus mechanisms for ensuring consistency are some of the most expensive operations in managing large amounts of data. Often, there is a trade off that involves reducing the coordination overhead at the price of accepting possible data loss or inconsistencies. As the demand for more efficient data centers increases, it is important to provide better ways of ensuring consistency without affecting performance.

In this paper we show that consensus (atomic broadcast) can be removed from the critical path of performance by moving it to hardware. As a proof of concept, we implement Zookeeper's atomic broadcast at the network level using an FPGA. Our design uses both TCP and an application specific network protocol. The design can be used to push more value into the network, e.g., by extending the functionality of middleboxes or adding inexpensive consensus to in-network processing nodes.

To illustrate how this hardware consensus can be used in practical systems, we have combined it with a main-memory key value store running on specialized microservers (built as well on FPGAs). This results in a distributed service similar to Zookeeper that exhibits high and stable performance. This work can be used as a blueprint for further specialized designs.

1 Introduction

Data centers face increasing demands in data sizes and workload complexity while operating under stricter efficiency requirements. To meet performance, scalability, and elasticity targets, services often run on hundreds to thousands of machines. At this scale, some form of coordination is needed to maintain consistency. However, coordination requires significant communication between instances, taking processing power away from the main task. The performance overhead and additional resources needed often lead to reducing consistency, resulting in less guarantees for users who must then build more complex applications to deal with potential inconsistencies.

The high price of consistency comes from the multiple rounds of communication required to reach agreement. Even in the absence of failures, a decision can be taken only as quickly as the network round-trip times allow it. Traditional networking stacks do not optimize for latency or specific communication patterns turning agreement protocols into a bottleneck. The first goal of this paper is to *explore whether the overhead of running agreement protocols can be reduced* to the point that it is no longer in the performance critical path. And while it is often possible to increase performance by "burning" more energy, the second goal is to *aim for a more efficient system*, i.e., do not increase energy consumption or resource footprint to speed up enforcing consistency.

In addition to the performance and efficiency considerations, there is an emerging opportunity for smarter networks. Several recent examples illustrate the benefits of pushing operations into the network [16, 41, 54] and using middleboxes to tailor it to applications [52, 9, 61, 49]. Building upon these advances, the following question arises: could agreement be made a property of the network rather than implementing it at the application level? Given the current trade off between complexity of operations and the achievable throughput of middleboxes, the third goal of this work is to *explore how to push down agreement protocols into the network in an efficient manner*.

Finally, data center architecture and the hardware used in a node within a data center is an important part of the problem. Network interface cards with programmable accelerators are already available from, e.g., Solarflare [55], but recent developments such as the HARP initiative from Intel [25] or the Catapult system of Microsoft [50] indicate that heterogeneous hardware is an increasingly feasible option for improving performance at low energy costs: the field programmable

[*] Part of this work was done while the author was at ETH Zürich.

gate arrays (FPGAs) used in these systems offer the opportunity of low energy consumption and do not suffer from some of the traditional limitations that conventional CPUs face in terms of data processing at line-rate. Catapult also demonstrates the benefits of having a secondary, specialized network connecting the accelerators directly among themselves. When thinking of agreement protocols that are bound by round trip times and jitter, such a low latency dedicated network seems quite promising in terms of efficiently reducing overhead. We do not argue for FPGAs as the only way to solve the problem, but given their increasing adoption in the data center, it makes sense to take advantage of the parallelism and performance/energy advantages they offer. This leads us to the fourth and final question the paper addresses: *can an FPGA with specialized networking be used to implement consensus while boosting performance and reducing overall overhead*?

Contribution. In this paper we tackle the four challenges discussed above: We implement a consensus protocol in hardware in order to remove the enforcement of consistency from the critical path of performance without adding more bulk to the data center. We create a reusable solution that can augment middleboxes or smart network hardware and works both with TCP/IP and application-specific network protocols using hardware and platforms that are starting to emerge. Therefore the solution we propose is both a basis for future research but also immediately applicable in existing data centers.

Results. In the paper we show how to implement Zookeeper's atomic broadcast (ZAB [43]) on an FPGA. We expose the ZAB module to the rest of the network through a fully functional low latency 10Gbps TCP stack. In addition to TCP/IP, the system supports an application-specific network protocol as well. This is used to show how the architecture we propose can be implemented with point-to-point connections to further reduce networking overhead. For a 3 node setup we demonstrate 3.9 million consensus rounds per second over the application specific network protocol and 2.5 million requests per second over TCP. This is a significant improvement over systems running on commodity networks and is on par even with the state of art systems running over lower latency and higher bandwidth Infiniband networks. Node to node latencies are in the microsecond range, without significant tail latencies. To illustrate how this hardware consensus can be used in practical systems, we have combined it with a main-memory key value store running on specialized microservers (built as well on FPGAs). This results in a distributed service similar to Zookeeper that exhibits a much higher and stable performance than related work and can be used as a blueprint for further specialized designs.

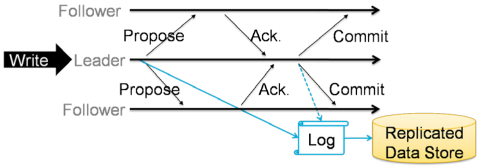


Figure 1: Zookeeper's Atomic Broadcast

2 Background

2.1 Zookeeper's Atomic Broadcast

There are many distributed systems that require some form of coordination for achieving their core services, and since implementing distributed consensus [34, 35] correctly is far from trivial [46], reusable libraries and solutions such as Zookeeper have emerged. Zookeeper is a centralized service that provides distributed synchronization, store configuration, and naming services for distributed systems. It achieves fault tolerance and high availability through replication.

At the core of Zookeeper is an atomic broadcast protocol (ZAB [33]) coupled with leader election that is used to ensure the consistency of modifications to the tree-based data store backing Zookeeper. ZAB is roughly equivalent to running Paxos [35], but is significantly easier to understand because it makes a simplifying assumption about the network. The communication channels are assumed to be lossless and strongly ordered (thus, Zookeeper in principle requires TCP).

We briefly describe the ZAB protocol in a 3 node setup (Figure 1): The atomic broadcast protocol of Zookeeper is driven by a leader, who is the only node that can initiate proposals. Once the followers receive proposals, they will acknowledge the receipt of these proposals thus signaling that they are ready to commit. When the leader received an acknowledgment from the majority of followers it will issue a commit message to apply the changes. Committed messages are persisted by default on a disk, but depending on the nature of the data stored in the service and failure scenarios, writing the log to memory can be enough. The order of messages is defined using monotonically increasing sequence numbers: the "Zxid'," incremented every time a new proposal is sent, and the "epoch" counter, which increases with each leader election round.

Zookeeper can run with two levels of consistency: strong [26] and relaxed (a form of prefix consistency [56]). In the strong case, when a client reads from a follower node, it will be forced to consult the leader whether it is up to date (using a *sync* operation), and if not, to fetch any outstanding messages. In the more relaxed case (no explicit synchronization on read) the node might return stale data. In the common case, however, its state mirrors the global state. Applications using Zookeeper often opt for relaxed consistency in order to increase read performance. Our goal is to make strong consistency cheaper and through this to deliver better value to the applications at a lower overall cost.

2.2 Reconfigurable Hardware

Field programmable gate arrays (FPGAs) are hardware chips that can be reprogrammed but act like application-specific integrated circuits (ASICs). They are appealing for implementing data processing operations because they allow for true dataflow execution [45, 57]. This computational paradigm is fundamentally different from CPUs in that all logic on the chip is active all the time, and the implemented "processor" is truly specialized to the given task. FPGAs are programmed using hardware description languages, but recently high level synthesis tools for compiling OpenCL [6], or domain specific languages [24] down to logic gates are becoming common.

FPGAs typically run at low clock frequencies (100-400 MHz) and have no caches in the traditional sense in front of the DDR memory. On the other hand the FPGA fabric contains thousands of on-chip block RAMs (BRAM) that can be combined to form different sized memories and lookup tables [13]. Recent chips have an aggregated BRAM capacity in the order of megabytes.

There are good examples of using FPGA-based devices in networks, e.g., smart NICs from Solarflare [55] that add an FPGA in the data path to process packets at line-rate, deep packet inspection [11] and line-rate encryption [51]. It has also been proposed to build middleboxes around FPGAs [9] because they allow for combining different functional blocks in a line-rate pipeline, and can also ensure isolation between different pipelines of different protocols. We see our work as a possible component in such designs that would allow middleboxes to organize themselves more reliably, or to provide consensus as a service to applications.

3 System Design

For prototyping we use a Xilinx VC709 Evaluation board. This board has 8 GB of DDR3 memory, four 10Gbps Ethernet ports, and a Virtex-7 VX690T FPGA (with 59 Mb Block RAM on chip). Our system consists of three parts: networking, atomic broadcast, and to help evaluate the implementation of the latter two, a key-value store as consensus application (Figure 2). The network stack was implemented using high level synthesis [60], the other two modules are written in Verilog and VHDL.

The Atomic Broadcast module replicates requests sent to the application (in our case the key-value store). Since it treats the actual requests as arbitrary binary data, it requires a thin header in front of them. The structure of the 16 B header is explained in Table 1: It consists of an operation "code" and ZAB-specific fields, such as epoch-number and Zxid. This is because the same header structure is used for communication between nodes and clients, and different node's atomic broadcast units. This means that not all messages will have a payload. As explained in Section 2.1, Zookeeper provides two levels of consistency, from which in our system we implement strong consistency by serving both reads and writes from the leader node. This setup simplifies the discussion and evaluation, however, serving strongly consistent read on followers is also possible.

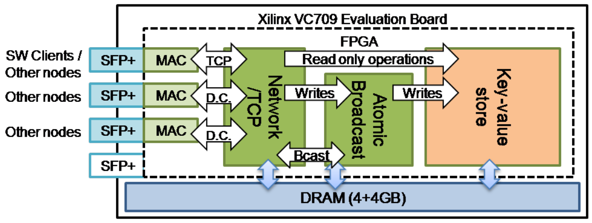


Figure 2: The target platform and system architecture

Bits	Description	Bits	Description
[15:0]	Magic number	[63:32]	Length of message
[23:16]	Sender Node ID	[95:64]	Zxid (req. sequence no.)
[31:24]	Operation code	[127:96]	Epoch number

Table 1: Structure of request header

When the atomic broadcast unit is used in conjunction with the key-value store, one can distinguish between two types of client requests: local ones (reads) and replicated ones (writes). Local requests are read operations that a node can serve from its local data store bypassing the atomic broadcast logic completely. Write requests need to be replicated because they change the global state. These replicated requests are "trapped" inside the atomic broadcast module until the protocol reaches a decision and only then are sent to the application, which will process them and return the responses to the client. For reaching consensus, the atomic broadcast module will send and receive multiple messages from other nodes. Since the atomic broadcast unit does not directly operate on the message contents, these are treated as binary data for the sake of replication.

4 Networking

The FPGA nodes implement two networking protocols: TCP/IP and an application specific one, used for point-to-point connections. As Figures 2 and 3 show, the network module connects to the Ethernet Network Interface provided by the FPGA vendor that handles Layer 1 and 2 (including MAC) processing before handing the packets to the IP Handler module. This module validates IP checksums and forwards packets to their protocol handlers. Additionally, data arriving from other FPGAs, using the application specific network protocol, shortcut

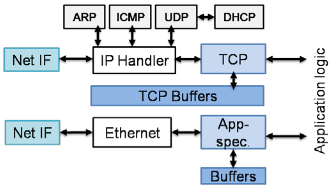


Figure 3: Network stack overview

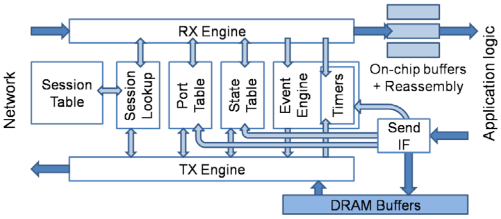


Figure 4: TCP stack details

the TCP/IP stack and are instead processed by an application specific network module. The ARP, ICMP and DHCP modules in Figure 3 provide the functionality necessary for integrating the FPGA into a real network.

4.1 TCP/IP in Hardware

The network stack is a modified version of earlier work [53]. The original design addresses the problem of very low connection counts in existing network stacks for FPGAs (typically in the tens or low hundreds). The changes introduced in this paper aim to further reduce the latency of the stack and reduce its memory footprint by tailoring the logic to the consensus and replication traffic.

The main benefit of using hardware for implementing a network stack is that it allows for building true dataflow pipelines and also for the isolation of send and receive paths so that they do not influence each other's performance negatively. Figure 4 shows the resulting architecture of the TCP stack in hardware: two pipelines that share connection state through the data structures. These data structures are: session lookup, port and state table, and an event engine backed by timers. The session lookup table contains the mapping of the 4-tuple (IP address and TCP port of destination and source) to session IDs, and is implemented as a content-addressable memory directly on the FPGA [32]. It holds up to 10k sessions in our current configuration. The port table tracks the state of each TCP port and the state table stores meta-information for each open TCP connection. The event engine is responsible for managing events and incoming requests from the Send Interface, and instructs the TX Engine accordingly.

4.2 Application-aware Receive Buffers

TCP operates on the abstraction of data streams, however data packets on the application level are usually very well defined. We take advantage of this application knowledge to reduce the latency of our network stack. The original version [53] of the network stack implemented "generic" receive buffers. In this version we replaced the DRAM buffer on the receive path with low latency on-chip BRAM. The smaller buffer space has no negative impact on throughput due to two reasons: 1) The application logic is designed to consume data at line-rate for most workloads, 2) In the datacenter TCP packets are in the common case rarely reordered [49]. Consequently, a smaller on-chip BRAM buffer will lower the latency without negatively impacting performance and frees up DRAM space for the consensus and application logic. Internally the BRAM buffers are organized as several FIFOs that are assigned dynamically to TCP sessions. By pushing down some knowledge about the application protocol (header fields), the BRAM buffers can determine when a complete request is available in a FIFO and then forward it to the application logic. In case all FIFOs fill up, we rely on TCP's built in retransmission mechanisms in order to not lose any data. For this reason on the transmit path a much larger buffer space is required, since packets have to be buffered until they are acknowledged. Therefore the high capacity DRAM buffer from our original design was kept.

4.3 Tailoring TCP to the Datacenter

TCP gives very strong guarantees to the application level, but is very conservative about the guarantees provided by the underlying network. Unlike the Internet, datacenter networks have well-defined topologies, capacities, and set of network devices. These properties, combined with knowledge about the application, allow us to tailor the TCP protocol and reduce the latency even further without giving up any of the guarantees provided by TCP.

Starting from the behavior of consensus applications and key-value stores we make two assumptions for the traffic of the key-value store and consensus logic to optimize the TCP implementation: a client request is always smaller than the default Ethernet MTU of 1500 B and clients are synchronous (only a single outstanding request per client). Additionally, we disable Nagle's algorithm which tries to accumulate as much payload from a TCP stream to fill an entire MTU. Since it waits for a fixed timeout for more data, every request small than MTU gets delayed by that timeout. The combination of disabling Nagle's algorithm, client requests fitting inside an MTU, and synchronous clients means that we can assume that in the common case and except for retransmission between the FPGAs, requests are not fragmented over multiple MTUs and each Ethernet frame holds a single request. Disabling Nagle's algorithm is quite common in software stacks through the TCP_NODELAY flag. Having our own hardware implementation we did an additional optimization to reduce latency by disabling Delayed Acknowledgments as described in RFC1122 [4], which says that a TCP implementation should delay an Acknowledgment for a fixed timeout such that it either can be merged with a second ACK message or with an outgoing payload message, thereby reducing the amount of bandwidth which is used for control messages. Since in our setup the network latencies between the FPGAs are in the range of a few microseconds, we decided to not just reduce the timeout but completely remove it. This way each message sent is immediately acknowledged. Obviously removing Delayed Acknowledgments and disabling Nagle's algorithm comes with the tradeoff that more bandwidth is used by control messages. Still, our experiments show that even for small messages we achieve a throughput of more than 7 Gbps considering only "useful" payload.

4.4 Application Specific Networking

In addition to the regular TCP/IP based channels, we have also developed a solution for connecting nodes to each other on dedicated links (direct connections, as we will refer to them in the paper, labeled D.C. in Figure 2), while remaining in-line with the reliability requirements (in-order delivery, retransmission on error). Packets are sent over Ethernet directly, and sequence numbers are the main mechanism of detecting data loss. These are inserted into requests where normally the ZAB-specific magic number is in the header – so the sequence number is actually increased with each logical request, not with each packet. Since the links are point-to-point, congestion control is not necessary beyond signaling backpressure (achieved through special control packets). If data was dropped due to insufficient buffer space on the receiving end, or because of corruption on the wire, the receiver can request retransmissions. To send and receive data over this protocol, the application uses special session numbers when communicating with the network stack, such that they are directly relayed to the application specific network module in Figure 3.

The design of the buffers follows the properties of the connections as explained above: the sending side maintains a single buffer (64 KB) per link from which it can resend packets if necessary, and the receiving side only reserves buffer space for request reassembly. Since the latency between nodes is in the order of microseconds, this buffer covers a time window of 50 μs on a 10 Gbps link, more than enough capacity for our purposes.

At the moment, our design only accommodates a limited number of nodes connected together with this protocol because there is no routing implemented and the FPGAs have only four Ethernet interfaces each. The Catapult platform [50] is a good example of what is possible over such secondary interconnects: it includes a 2D torus structure where FPGAs route packets over the dedicated network, while using a simple application-specific protocol. We plan to eventually evaluate our system at larger scale using such a network interconnect.

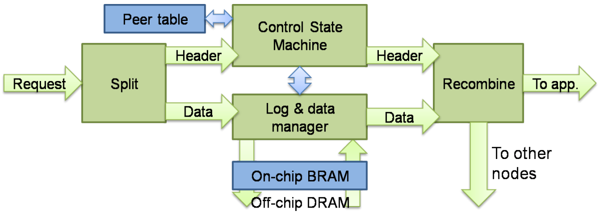


Figure 5: Overview of the atomic broadcast module

5 Atomic Broadcast in Hardware

The overall benefit of using hardware for implementing consensus is that nodes have predictable performance, thereby allowing the protocol to function in the"best case scenario" most of the time. Latencies are bound and predictable, so with careful adjustments of on-chip buffers and memories, the hardware solution can for instance avoid in most cases to access the log in DRAM and read the cached head from on chip memory instead. Even the "less common" paths in the algorithm can perform well due to the inherent parallelism of FPGA resources, and the ability to hide memory access latencies through pipelining for instance. Another example is the timeout used for leader election that is much lower than what would be feasible in software solutions. In conclusion, the high determinism of hardware, low latency and inherent pipeline parallelism are a good fit for ZAB and there was no need to write a new solution from scratch.

By design, the atomic broadcast logic treats the data associated with requests as arbitrary binary data. This decouples it from the application that runs on top. For the purpose of evaluation, in this paper we use a key-value store but integrating other applications would be straightforward as well.

Inside the consensus module the control and data planes are separated, and the Control State Machine and the Log/Data Manager shown in Figure 5 can work in parallel to reduce latencies more easily. There are two additional blocks in the figure to complete the consensus functionality. The Splitter splits the incoming requests into command word and payload, and the Recombine unites commands with payloads for output. Headers (i.e., command words) are extracted from requests and reformatted into a single 128 bit wide word, so that they can be manipulated and transmitted internally in a single clock cycle (as compared to two on the 10Gbps data bus that is 64 bits wide). Similarly, payload data is aggregated into 512 bit words to match the memory interface width. When the control state machine (controller) issues a command (header) that has no encapsulated data,

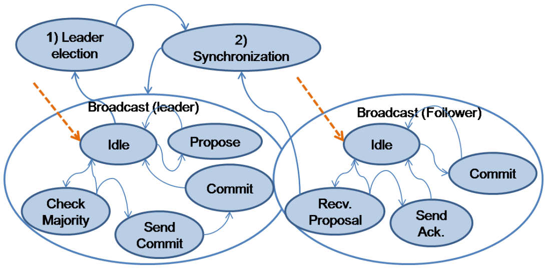


Figure 6: Abstract states used for implementing the Zookeeper Atomic Broadcast in the FPGA

Operation/State	**Cost** (clock cycles)	**Max for:** **3 nodes**	**Max for:** **7 nodes**
L_1 Create and send proposal	2+C_{nodes}x2	19.5 M/s	9.75 M/s
F_1 Recv. proposal and send acknowledgment	2	78 M/s	78 M/s
L_2 Recv. acknowledgment and check majority	2+C_{nodes}+L_{cLog}	17.3 M/s	13 M/s
F_2 Commit	1+L_{cLog}	39 M/s	39 M/s
L_3 Commit	3+C_{nodes}	26 M/s	15.6 M/s
Consensus round (leader)	$L_1 + L_2 + L_3$	7.1 M/s	4.1 M/s
Consensus round (follower)	$F_1 + F_2$	26 M/s	26 M/s

Table 2: Cost of ZAB operations and the theoretical maximum consensus rounds per second over 10GbE

such as the *acknowledgment* or *commit* messages of the ZAB protocol, this passes through the Recombine module without fetching a payload from the log. If, on the other hand, there is encapsulated data to be sent then the controller will request the data in parallel to creating the header. The system is pipelined, so it is possible to have multiple outstanding requests for data from memory.

5.1 State Machine

The state machine that controls the atomic broadcast is based on the ZAB protocol as described in [43]. Figure 6 shows the "super states" in which the state machine can be (each such oval on in the figure hides several states of its own). Transitions between states are triggered by either incoming packets or by means of event timers. These timers can be used for instance to implement timeouts used in leader election, detection of failed nodes, etc., and operate in the range of tens of μs.

Table 2 shows an overview of how many clock cycles the more important states of the state machine take to be processed. Most of them are linear in cost with the number of nodes in a setup (C_{nodes}), or the time it takes to seek in the command log as the parameter L_{cLog} (3 cycles in the average case in our current design). The theoretical maximum throughput achievable by the control unit shown in Table 2 for the 3 node setup we use in the Evaluation is higher than the maximum throughput in reality as our system is limited by 10Gbps networking most of the time. If we wanted to scale our system up to 40 Gbps networking, this component could be clocked up to 300 MHz (independently from the rest of the pipeline) and then it would have enough performance to handle the increased message rate. The rest of the logic inside the atomic broadcast module handles the payloads only, and these paths could be made wider for 4 x throughput.

On each node there is a table to hold the state of the other nodes in the cluster. The table resides in BRAM and in the current implementation holds up to 64 entries. Each entry consists of information associated with Zookeeper atomic broadcast (Zxid, epoch, last acknowledged Zxid, etc.), a handle to the session opened to the node, and a timestamp of the node's last seen activity. Since the mapping from session number to network protocol or even network interface is made in the networking module, the controller is independent of the network details and works the same for messages received over any protocol or connection.

5.2 Quorums and Committing

Zookeeper atomic broadcast allows the pipelining of requests, so when the leader's controller receives a client request that needs to be replicated it will send out the proposal and mark its Zxid as the highest that has already been sent but not acknowledged or committed. When an acknowledgment is received from another node, the leader's controller will test if a quorum (majority) has been reached on that Zxid. This is done by iterating through the active nodes in the state table: if enough nodes have already acknowledged, the leader's controller will send out commit messages to all nodes that already acknowledged the proposal. Then the leader will instruct the log unit to mark the operation as successful and to return the payload so that the application can process the original request. On the follower, the receipt of a commit message will result in the same operations of updating the log and preparing the data for the application. In case a node sends its acknowledgment after the operation has already been committed, the leader will issue a commit to that node as a response.

The system offers tunable consistency by allowing the quorum-checking function to be updated at runtime. To be more specific, one can change between either waiting for a majority or waiting for all nodes to respond. The latter behavior could be useful in cases when failures of nodes are assumed to be transient, but updates have to happen absolutely at the same time on all nodes (like changing a policy on a set of middleboxes). While in software this could lead to much higher response times, in the Evaluation section we show the benefits of the low latency hardware.

5.3 Maintaining a Log

After the payload is separated from the command it is handed to the Log and Data Manager (bottom half of Figure 5). The payload is added to an append-only log, and read out later to be sent to other nodes with the proposal commands. When an operation commits, this event is also added to the log (a physically different location, reserved for command words) with a pointer to the payload's location. In case a synchronization has to happen, or the node needs to read its log for any other reason, the access starts with the part of the log-space that contains the commands. Since each entry is of fixed size, it is trivial to search a command with a given Zxid.

To reduce the latency of accessing this data structure we keep the most recent entries in on-chip BRAM, which is spilled to DRAM memory in large increments. Of course if the payloads are large only a small number will fit into the first part of the log. However, this is not really an issue because in the common case each payload is written once to the log when received and then read out immediately for sending the proposals (it will still be in BRAM at this point), and then read again later when the operation commits. The aspect where the atomic broadcast unit and the application need to work together is log compaction. In the case of the key-value store, the log can be compacted up to the point where data has been written to the key-value store, and the key-value store notifies the atomic broadcast unit of each successful operation when returning the answer to the client.

Our design is modular, so that the log manager's implementation could change without requiring modifications in the other modules. This is particularly important if one would want to add an SSD to the node for persistent log storage. We have mechanisms to extend the FPGA design with a SATA driver to provide direct access to a local SSD [59]. Although we have not done it in the current prototype, this is part of future work as part of developing a data appliance in hardware. Alternatively, one can use battery backed memory [5], which in the context of the FPGA is a feasible and simpler option.

5.4 Synchronization

When a node fails, or its network experienced outages for a while it will need to recover the lost messages from the leader. This is done using *sync* messages in the ZAB protocol. In our implementation, when a follower detects that is behind the leader it will issue a *sync* message. The leader will stream the missing operations from the log to the follower. These messages will be sent with an opcode that will trigger their immediate commit on the other end. In the current design the leader performs this operation in a blocking manner, where it will not accept new input while sending this data. It is conceivable to perform this task on the side, but for simplicity we implemented it this way for this prototype design.

If for some case the other node would be too far behind the leader, and syncing the whole log would take longer than copying the whole data structure in the key-value store (or the log has already been compacted beyond the requested point) there is the option of *state transfer* at bulk: copying the whole hash table over and then sending only the part of the log that has been added to the table since the copy has begun. Of course this tradeoff depends on the size of the hash table, and is an experiment that we defer to our future work.

5.5 Bootstrapping and Leader Election

On initial start-up each node is assigned a unique ID by an outside entity, e.g., a configuration master [36]. This ID is a sequence number that is increased as nodes are added. If a node joins after the initial setup, it gets the next available ID and all other nodes are notified. If a node fails and then recovers, it keeps its ID. When thinking of smart network devices or middleboxes connected together in a coordination domain, it is reasonable to expect much less churn than with regular software nodes.

We implement the leader election following the algorithm in the ZAB paper [33], with the optimization that the followers will propose prospective leaders in a round-robin fashion, i.e., proposing the next higher ID once the current leader is unreachable. Nodes transition to leader election once no message or heartbeat has been received from the leader for a given timeout (based on the maximum consensus time in our use-case we set this to $50\mu s$). We perform the synchronization phase after the leader election (discovery phase in the ZAB paper) in a pull-based manner. This means that the newly elected leader will explicitly ask the most up to date follower to send it the requests with which it might be behind instead of followers actively sending their histories. Requests arriving from clients during leader election and synchronization will be dropped by default, to allow the clients to reconfigure based on a timeout mechanism. One simple optimization that we implement is responding to requests arriving during the leader election with a message that will prompt the client to switch over to the next leader directly without timeouts. Further, more sophisticated optimizations are possible, but are deferred to future work.

6 Use-case: Key-value Store

In order to test the atomic broadcast unit with a realistic application running on the same FPGA we implemented a key-value store that at its core uses the hash table design from our earlier work [29]. It is compatible with memcached's ASCII protocol and implements the *set*, *get*, *delete* and *flush all* commands. In addition, it supports memcached's check-and-set (*cas*) as well. The design is aggressively pipelined and handles mixed workloads well. As we previously demonstrated, the internal pipeline can process more than 11 million memcached requests per second, enough to saturate a 10Gbps connection even with keys and values as small as 16 B.

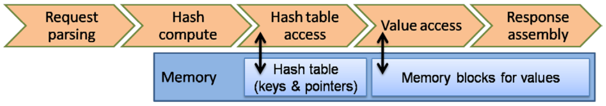


Figure 7: Internal pipeline of the key-value store

	32B Request	512B Req.	1KB Req.
Ethernet loopback	0.6μs	1.4μs	2.2μs
TCP loopback	1.5μs	3.8μs	6.5μs
Direct Conn. loopback	0.7μs	1.5μs	2.3μs
DRAM access latency	0.2μs		
Ping from client	15μs	35μs	–

Table 3: Latencies of different components of the system

Keys are hashed with a hash function that is based on multiplication and shifting and processes the input one byte at a time (similar to a Knuth's hash function). To meet the line-rate requirements we rely on the parallelism of the FPGA and create multiple hashing cores that process the keys in the order of arrival. We solve collisions by chaining, and the first four entries of every chain are pre-allocated and stored in memory in such a way that they can be read in parallel. This is achieved by dividing each physical memory line (512b on our platform) into four equal parts belonging to different entries, and tiling keys over multiple physical memory addresses. The start address of each bucket is at a multiple of 8 memory lines, which allows for keys of up to 112 B long (the header of each entry is 16 B) to be stored. This size should be enough for most common use-cases [10].

In order to hide the memory latency when accessing the data structures in off-chip DDR memory, the hash table is implemented as series of pipelined stages itself. Multiple read commands can be issued to the memory, and the requests will be buffered while waiting for the data from memory. While with this concurrency there is no need for locking the hash table entries in the traditional sense, the pipeline has a small buffer on-chip that stores in-flight modifications to memory lines. This is necessary to counter so called read-after-write hazards, that is, to make sure that all requests see a consistent state of the memory. A benefit of no locking in the software sense, and also hiding memory latency through pipelining instead of caching, is that the hash table is agnostic to access skew. This is an improvement over software because in the presence of skew any parallel hash table will eventually become bottlenecked on a single core.

Similarly to the related work in software [7] and hardware [30], we store the values in a separate data structure from keys. This allows for more flexible memory allocation strategies, and also the option to provide more complex ways of managing memory in the future without modifying the hash table data structure. At the moment we use a simple block based memory allocation scheme that allocates memory linearly. When a key is inserted into the hash table, and its value placed in memory, its slot in the value store is reused for subsequent updates as long as the modified value is smaller than or equal in size to the previous one.

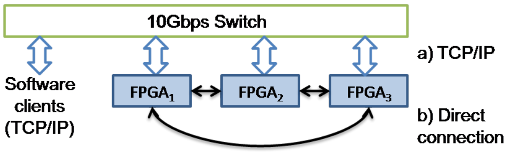


Figure 8: Evaluation setup of our prototype system

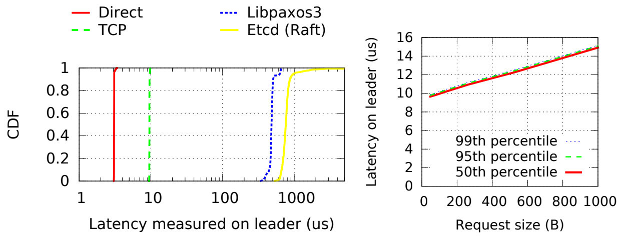
7 Evaluation

7.1 Setup

For evaluation, we use 12 machines connected to a 10Gbps 48 port Cisco Nexus 5596UP switch and three FPGAs connected to the same switch (Figure 8). FPGAs communicate either over TCP or the specialized network, i.e., direct connections. The three node setup mirrors the basic fault-tolerant deployment of Zookeeper that can tolerate one faulty node. The client machines have dual-socket Xeon E5-2609 CPUs, with a total of 8 cores running at 2.4 GHz, 128 GB of memory and an Intel 82599ES 10Gbps network adapter. The machines are running Debian Linux (jessie/sid with kernel version 3.12.18) and use the standard ixgbe drivers. Our load generator was memaslap [2] with modifications to include our ZAB header in the requests.

7.2 Baselines

The performance of consensus protocols is sensitive to latency, so we performed a series of micro-benchmarks and modeling to determine the minimal latencies of different components of our system with differently sized packets. As the results in Table 3 show, the transmission of data through TCP adds the most latency (ranging between 1 and 7μs), but this is expected and is explained by the fact that the packet goes through additional checksumming and is written and read from a DRAM buffer. An other important result in Table 3 is that round trip times of ping messages from software are almost an order of magnitude higher than inter-FPGA transmission times, which highlights the shortcomings of the standard networking stack in software. The measurements were taken using the ping-flood command in Linux. In the light of this, we will mainly report consensus latencies as measured on the leader FPGA (we do this by inserting two timestamps in the header: one when a message is received and the other when the first byte of the response is sent), and show times measured on the client for experiments that involve the key-value store more.



(a) Small requests, related work (b) Increasing size over TCP

Figure 9: Consensus round latency on leader

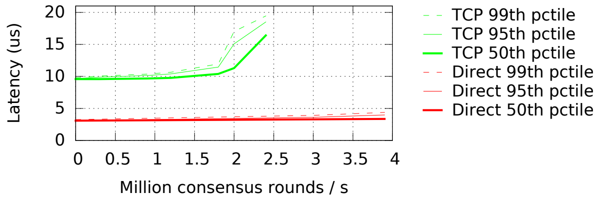


Figure 10: Load vs. Latency on leader

7.3 Cost of Consensus

Systems such as Zookeeper require hundreds of microseconds to perform a consensus round [48, 20] even without writing data to disk. This is a significant overhead that will affect the larger system immediately, and here we explain and quantify the benefits of using hardware for this task. Instead of testing the atomic broadcast module in isolation, we measure it together with the application on the chip, using memaslap on a single thread sending one million consecutive write requests to the key-value store that need to be replicated. We chose very small request size (16 B key and 16 B value) to ensure that measurements are not influenced by the key-value store and stress mostly the atomic broadcast module.

Figure 9(a) depicts the probability distribution of a consensus round as measured on the leader both when using TCP and direct connections to communicate with followers. Clearly, the application-specific network protocol has advantages over TCP, reducing latencies by a factor of 3, but the latter is more general and needs no extra infrastructure. Figure 9(b) shows that the latency of consensus rounds increase only linearly with the request size, and even for 1KB requests stay below 16μs on TCP. To put the hardware numbers in perspective we include measurements of Libpaxos3 [3] and the Raft implementation used in Etcd [1]. We instrumented the code of both to measure the latency of consensus directly on th leader and deployed them on three nodes in our cluster. Unsurprisingly the software solutions show more than an order of magnitude difference in average latency, and have significantly higher 99th percentiles even for this experiment where the system handles one request at a time.

Figure 10 shows how the FPGA system fares under increasing load of replicated requests. As later shown in the experiments, with small payloads (<48 B) the system can reach up to 2.4 million consensus rounds per second over TCP and almost 4 million over direct connections before hitting bottlenecks. In our graph the latencies do not increase to infinity because we increased throughput only until the point where the pipeline of the consensus module and the input buffers for TCP could handle load without filling all buffers, and the clients did not need to retransmit many messages. Since we measured latency at the leader, these retransmitted messages would lead to false measurements from the leader's perspective.

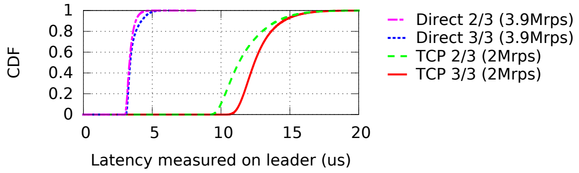


Figure 11: Latency of consensus on leader when waiting for majority or all nodes, on TCP and direct connections

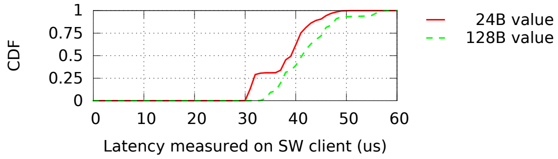


Figure 12: Round-trip times with replication measured on client

7.4 Quorum Variations

The leader can be configured at runtime to consider a message replicated either when a majority of nodes acknowledged or all of them. The second variant leads to a system of much stronger consistency, but might reduce availability significantly. We measured the effect of the two strategies on consensus latency, and found that even when the system is under load waiting for an additional node does not increase latencies significantly. This is depicted in Figure 11 both for TCP and direct, "2/3" being the first strategy committing when at least two nodes agree and "3/3" the second strategy when all of them have to agree before responding to the client.

7.5 Distributed Key-value Store

The rest of the evaluation looks at the atomic broadcast module as part of the distributed key-value store running on the FPGAs. We measure the round trip times (latency) on the clients and report maximum throughput with multiple client machines. As seen in Figure 12, for a single threaded client round trip times are 30μs larger than the measurements taken on the leader. The reason for this is the inefficiency of the software network stack, and is in line with the ping micro-benchmarks. Interestingly, even though these numbers are measured on a system with a single client and one TCP connection, software still adds uncertainty to the high percentiles.

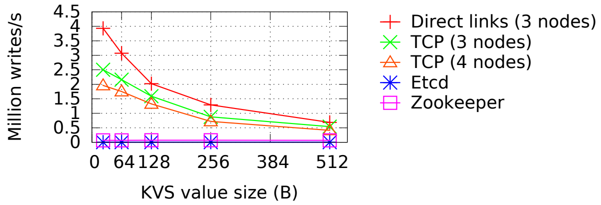


Figure 13: Consensus rounds (writes operations) per second as a function of request size

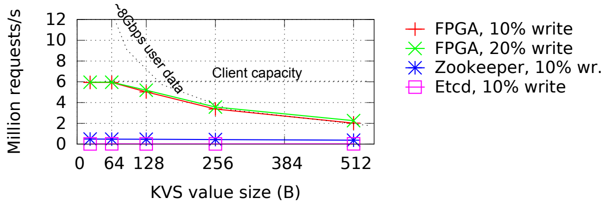


Figure 14: Mixed workload throughput measured on leader without and with replication (3 nodes)

When varying the size of the values in the write requests to the key-value store, we can exercise different parts of the system. The key size is kept at a constant 16 B for all experiments. Figure 13 and Figure 14 show the achievable throughput for write-only and mixed workloads (for the former all requests are replicated, for the latter only a percentage). Naturally, the write-only performance of the system is limited by the consensus logic when using direct connections, and by the combination of the consensus logic and the TCP/IP stack otherwise. This is because the transmit path on the leader has to handle three times as many messages as the receive path, and random accesses to DRAM limit performance. To further explore this aspect we ran experiments with a 4 node FPGA setup as well, and seen that the performance scales as expected.

For mixed workloads and small requests the clients' load generation capability is the bottleneck, while for larger messages performance is bound by the network capacity. This is illustrated by the two workloads in Figure 14, one with 10% writes and the other with 20%. Since they show the same performance, the atomic broadcast logic is not the bottleneck in these instances. The workload with 20% writes is actually slightly faster because the average size of responses to the clients gets smaller (each read request to the key-value store will return the key, value and headers, while write requests only return headers and a success message).

We ran Zookeeper and Etcd on three machines each and performed the same mixed-workload experiment as before. To make sure that they are not impacted by hard drive access times, we set up ram disks for persisting logs. Figure 14 shows that their performance is not bound by the network, and is mostly constant even for large messages. Both are almost an order of magnitude slower for small messages than hardware.

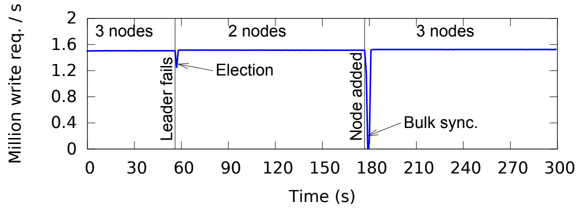


Figure 15: Leader election triggered while client issue an all-write workload, nodes connected via TCP links

7.6 Leader Election

To exercise leader election and recovery we simulate a node failure of the leader, which results in a leader election round without significant state transfer since the FPGAs do not drift much apart under standard operation. Hence leader election over TCP/IP takes approximately as long as a consensus round (10μs in average), not counting the initial timeout of the followers (50μs) Figure 15 depicts the experiment: we generate write-only load from several clients for a three node FPGA cluster communicating over TCP and at the 56 s mark the leader node fails and a new leader is elected. To make client transition possible we modified memaslap and added a timeout of 100 ms before trying an other node (the clients retry in the same round robin order in which the FPGAs try to elect leaders). The graph indicates that the dip in performance is due to the 100 ms inactivity of clients, since leader election takes orders of magnitude less.

Synchronization of state between nodes happens for instance when a new node joins the cluster. In Figure 15 we shows the previously failed node recovering after 2 minutes and prompting the new leader to synchronize. Since at this point the log has been compacted, the leader will bulk transfer the application state that consists of the hash table and value area, occupying 256MB and 2GB, respectively. During synchronization the leader will not handle new write requests to keep the state stable, hence the clients will repeatedly time out and resume normal operation only once the leader has finished the state transfer. The results show that, as expected, this step takes between 2-3 seconds, the time necessary to send the state over 10Gbps network plus clients resuming.

This experiment allows us to make two observations. First, leader election can be performed very quickly in hardware because detecting failed nodes happens in shorter time frames than in software (i.e., in order of 10s of μs). Hence, leader-change decisions can be taken quickly thanks to low round-trip times among nodes. Second, the cost of performing bulk transfers shows that in future work it will be important to optimize this operation. The hardware could benefit from the approach used by related work, such as DARE [48], where the followers synchronize the newly added node. This leads to smaller performance penalty incurred by state transfer at the cost of a slightly more complex synchronization phase.

Component	LUTs	BRAM	DSPs
PHY and Ethernet (x3)	16k	18	0
TCP/IP + App-spec.	23k	325	0
Memory interface	47k	127	0
Atomic broadcast	10k	340	0
Key-value store	28k	113	62
Total used *(% of available)*	124k *(28%)*	923 *(63%)*	62 *(2%)*

Table 4: Detailed breakdown of the resource usage

7.7 Logic and Energy Footprint

To decide whether the hardware consensus protocol could be included in other FPGA-based accelerators or middleboxes we need to look at its logic footprint. FPGAs are a combination of look-up tables (LUTs) that implement the "logic", BRAMs for storage, and digital signal processors (DSPs) for complex mathematical operations. In general, the smaller the footprint of a module, the more additional logic can fit on the chip besides it. Table 4 shows that the ZAB module is smaller than the network stack or the key-value store, and uses only a fraction of the LUTs on the chip. The resource table also highlights that a big part of the BRAMs are allocated for networking-related buffers and for log management. While it is true that some of these could be shrunk, in our current setup where there is nothing else running on the FPGA, we were not aiming at minimizing them.

One of the goals of this work is to show that it is possible to build a Zookeeper-like service in an energy efficient way. The power consumption of the FPGAs, even when fully loaded, is 25 W – almost an order of magnitude lower than the power consumption of a x86 server.

8 Related Work

8.1 Coordination in Systems

Paxos [34, 35] is a family of protocols for reaching consensus among a set of distributed processes that may experience failures of different kinds, including ones in the communication channels (failure, reordering, multiple transmission, etc.). While Paxos is proven to be correct, it is relatively complex and difficult to implement, which has led to alternatives like Raft [46], ZAB [27, 43] or chain replication [58]. There is also work on adapting consensus protocols for systems that span multiple physical datacenters [38, 40, 15], and while they address difficult challenges, these are not the same problems faced in a single data-center and tight clusters.

Paxos and related protocols are often packaged as coordination services when exposed to large systems. Zookeeper [27] is one such coordination service. It is a complex multi-threaded application and since its aim is to be as universal as possible, it does not optimize for either the network or the processor. Related work [48, 20] and our benchmarks show that its performance is capped around sixty thousand consensus rounds per second and that its response time is at least an order of magnitude larger than the FPGA (300-400μs using ram disks). Etcd [1], a system similar to Zookeeper, written in Go and using Raft [46] at its core has lower throughput than Zookeeper. This is partially due to using the HTTP protocol for all communication (both consensus and client requests) which introduces additional overhead.

Many systems (including e.g., the Hadoop ecosystem) are based on open source coordination services such as Zookeeper and Etcd, or proprietary ones (e.g., the Chubby [14] lock server). All of them can benefit from a faster consensus mechanisms. As an illustration, Hybris [20] is a federated data storage system that combines different cloud storage services into a reliable multi-cloud system. It relies on Zookeeper to keep metadata consistent. This means that most operations performed in Hybris directly depend on the speed at which Zookeeper can answer requests.

8.2 Speeding up Consensus

Recently, there has been a high interest in speeding up consensus using modern networking hardware or remote memory access. For instance DARE [48] is a system for state machine replication built on top of a protocol similar to Raft and optimized for one-sided RDMA. Their 5 node setup demonstrates very low consensus latency of $<15\mu$s and handles 0.5-0.75 million consensus rounds per second. These numbers are similar to our results measured on the leader for 3 nodes (3-10μs) and, not surprisingly, lower than those measured on the unoptimized software clients. While this system certainly proves that it is possible to achieve low latency consensus over Infiniband networks and explores the interesting idea of consensus protocols built on top of RDMA, our hardware-based design achieves higher throughput already on commodity 10 GbE and TCP/IP.

FaRM [23] is a distributed main-memory key value store with strong consensus for replication and designed for remote memory access over 40Gbps Ethernet and Infiniband. It explores design trade-offs and optimizations for one-sided memory operations and it demonstrates very high scalability and also high throughput for mixed workloads (up to 10M requests/s per node). FaRM uses a replication factor of three for most experiments and our hardware solution performs comparably both in terms of key-value store performance (the hardware hash table reaches 10 Gbps line-rate for most workloads [29]) and also in terms of consensus rounds per second, even though the FPGA version is running on a slower network.

NetPaxos [18] is a prototype implementation of Paxos at the network level. It consists of a set of OpenFlow extensions implementing Paxos on SDN switches; it also offers an alternative, optimistic protocol which can be implemented without changes to the Open- Flow API that relies on the network for message ordering in low traffic situations. Best case scenarios for NetPaxos exhibit two orders of magnitude higher latency than our system, FARM, or DARE. It can also sustain much lower throughput (60k requests/s). The authors point out that actual implementations will have additional overheads. This seems to indicate that it is not enough to push consensus into the network but it is also necessary to optimize the network and focus on latency to achieve good performance. In more recent work [17] the same authors explore and extend P4 to implement Paxos in switches. While P4 enables implementing complex functionality in network devices, the high level of abstraction it provides might make it difficult to implement the kind of protocol optimizations we describe in this paper and that are necessary to achieve performance comparable to that of conventional systems running over Infiniband.

Similar to the previously mentioned work, Speculative Paxos[49] suggests to push certain functionality into the network, e.g., message ordering. The design relies on specific datacenter characteristics, such as the structured topology, high reliability and extensibility of the network through SDN. Thereby, it could execute requests speculatively and synchronization between replicas only has to occur periodically. Simulations of the proposed design show that with increasing number of out-of-order messages the throughput starts to decrease quickly, since the protocol and application have to rollback transactions.

8.3 Quicker and Specialized Networking

One of the big challenges for software applications facing the network is that a significant time is spent in the OS layers of the network stack [47, 31] and on multicore architectures response times can increase as a result of context switching and memory copies from the NIC to the right CPU core. As a result, there are multiple frameworks for user-space networking [28, 31], and on the other end of the spectrum, operating systems [12, 47] that aim to speed up networking by separating scheduling and management tasks. The use of RDMA [22, 44] is also becoming common to alleviate current bottlenecks, but there are many (legacy) systems that rely on the guarantees provided by TCP, such as congestion control, inorder delivery and reliable transmission. Although some functionality of the network stack is offloaded to the NIC, processing TCP packets still consumes significant compute resources at the expense of the applications. Hardware systems, as we present in this paper, are implementing network processing as a dataflow pipeline and thereby can provide very high performance combined with the robustness and features of TCP.

A good example of what can be achieved with userspace networking is MICA [37], a key-value store built from the ground up using Intels DPDK [28] library. The results of this work are very promising: when using a minimalistic stateless protocol the complete system demonstrates over 70 million requests per second over more than 66Gbps network bandwidth (using a total of 8 network interfaces and 16 cores). It is important to note however that in MICA and similar systems skewed workloads will experience slowdowns due to the partitioned nature of the data structures. Additionally, a significant part of the servers logic (for instance computing the hash function on keys, or load balancing) is offloaded to clients. Our aim with the hardware solution on the other hand was to offer high throughput, low latency while relying on simple clients and commodity networks.

8.4 Hardware for Middleboxes

There is a wide spectrum of middlebox implementations ranging from all-software [42, 21, 8, 39], through hybrid [52, 9], to all-hardware [19]. One advantage of using FPGA-based solutions over software is that data can be processed at line-rate and only a minimal overhead in terms of latency is added. In ClickOS [42], for instance, adding a 40ms delay to get load balancing or congestion control is considered a good tradeoff. A hardware-based solution like the one we propose can perform even more complex operations, possibly involving coordination and consensus, in a fraction of that overhead.

9 Conclusion

In this paper we have explored a number of research questions aiming at determining whether the overhead of consensus can be removed as a bottleneck in distributed data processing systems. First, we have shown that it is possible to reduce the cost of reaching consensus without compromising reliability or correctness, through the means of specialized hardware. Second, based on the low latency and high throughput achieved, we have shown how to use the hardware consensus to implement a fully functional version of Zookeeper atomic broadcast with a corresponding key-value store. Third, we have argued that the proposed consensus module is agnostic to the actual request contents sent to the application and, hence, it could easily be integrated with middleboxes or other accelerators/microservers built with FPGAs. Finally, we have explored the benefits of using a custom messaging protocol for reducing latency, establishing the basis for further research into application specific protocols over secondary networks.

Acknowledgments

We would like to thank Xilinx for the donation of the FPGA boards used in this paper. The work reported in this paper is partially funded by Xilinx, Microsoft Research and the Swiss State Secretariat for Education Research and Innovation (SERI) under contract number 15.0025 (as part of the SUPERCLOUD H2020 project). We would also like to thank our shepherd, Wyatt Lloyd, for his help in improving our paper.

References

[1] Etcd repository in the CoreOS project. https://github.com/coreos/etcd.

[2] Libmemcached-1.0.18. https://launchpad.net/libmemcached/.

[3] LibPaxos3 repository. https://bitbucket.org/sciascid/libpaxos.

[4] Network working group: Requirements for internet hosts – communication layers. https://tools.ietf.org/html/rfc1122.

[5] Viking Technology. http://www.vikingtechnology.com/.

[6] Altera. Programming FPGAs with OpenCL. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01173-opencl.pdf.

[7] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP'09*.

[8] James W Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xOMB: extensible open middleboxes with commodity servers. In *ANCS'12*.

[9] Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, and Nick Feamster. Switchblade: a platform for rapid deployment of network protocols on programmable hardware. *ACM SIGCOMM CCR*, 40(4), August 2010.

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS'12*.

[11] Michael Attig and Gordon Brebner. 400 Gb/s programmable packet parsing on a single FPGA. In *ANCS'11*.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *OSDI'14*.

[13] Stephen Brown and Jonathan Rose. FPGA and CPLD architectures: A tutorial. *IEEE Design & Test*, 13(2), June 1996.

[14] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI'06*.

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Googles globally distributed database. *ACM TOCS*, 31(3), August 2013.

[16] Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. NaaS: Network-as-a-Service in the Cloud. In *Hot-ICE'12*.

[17] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM CCR*, 2016.

[18] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at network speed. In *SOSR'15*.

[19] Lorenzo De Carli, Yi Pan, Amit Kumar, Cristian Estan, and Karthikeyan Sankaralingam. Plug: flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *SIGCOMM'09*.

[20] Dan Dobre, Paolo Viotti, and Marko Vukolić. Hybris: Robust hybrid cloud storage. In *SOCC'14*.

[21] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *SIGOPS'09*.

[22] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *NSDI'14*.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP'15*.

[24] Nivia George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *FPL'14*.

[25] PK Gupta. Xeon+FPGA platform for the data center. In *CARL'15*.

[26] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), July 1990.

[27] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC'10*.

[28] Intel. DPDK networking library. http://dpdk.org/.

[29] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. A hash table for line-rate data processing. *ACM TRETS*, 8(2), March 2015.

[30] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees A. Vissers. A flexible hash table design for 10Gbps key-value stores on FPGAs. In *FPL'13*.

[31] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *NSDI'14*.

[32] Weirong Jiang. Scalable ternary content addressable memory implementation using FPGAs. In *ANCS'13*.

[33] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN'11*.

[34] Jonathan Kirsch and Yair Amir. Paxos for system builders: An overview. In *LADIS '08*, 2008.

[35] Leslie Lamport. Generalized consensus and paxos. Technical report, Microsoft Research MSR-TR-2005-33, 2005.

[36] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *PODC'09*.

[37] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: a holistic approach to fast in-memory key-value storage. In *NSDI'14*.

[38] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI'13*.

[39] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. ServerSwitch: A programmable and high performance platform for data center networks. In *NSDI'11*.

[40] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *PVLDB*, 6(9), July 2013.

[41] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *CoNEXT'14*.

[42] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *NSDI'14*.

[43] André Medeiros. Zookeeper's atomic broadcast protocol: theory and practice. Technical report, 2012.

[44] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *USENIX ATC'13*.

[45] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on FPGAs. *PVLDB*, 2(1), August 2009.

[46] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC'14*.

[47] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *OSDI 14*.

[48] Marius Poke and Torsten Hoefler. DARE: high-performance state machine replication on RDMA networks. In *HPDC'15*.

[49] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *NSDI'15*.

[50] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jordan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA'14*.

[51] Safenet. Ethernet encryption for data in motion. `http://www.safenet-inc.com/data-encryption/network-encryption/ethernet-encryption/`.

[52] Alan Shieh, Srikanth Kandula, and Emin Gun Sirer. Sidecar: building programmable datacenter networks without programmable switches. In *HotNets'10*.

[53] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware. In *FCCM'15*.

[54] Solarflare. Accelerating memcached using Solarflare's Flareon Ultra server I/O adapter. December 2014. `http://www.http://solarflare.com/Media/Default/PDFs/Solutions/Solarflare-Accelerating-Memcached-Using-Flareon-Ultra-server-IO-adapter.pdf`.

[55] Solarflare. Application OnLoad Engine (AOE). `http://www.solarflare.com/applicationonload-engine`.

[56] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12), December 2013.

[57] Jens Teubner and Louis Woods. Data processing on FPGAs. *Morgan & Claypool Synthesis Lectures on Data Management*, 2013.

[58] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI'04*.

[59] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex - an intelligent storage engine with support for advanced SQL off-loading. *PVLDB*, 7(11), July 2014.

[60] Xilinx. Vivado HLS. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[61] Wei Zhang, Timothy Wood, KK Ramakrishnan, and Jinho Hwang. Smartswitch: Blurring the line between network infrastructure & cloud applications. In *HotCloud'14*.

StreamScope: Continuous Reliable Distributed Processing of Big Data Streams

Wei Lin* Microsoft
Haochuan Fan* Microsoft
Zhengping Qian* Microsoft Research
Junwei Xu Microsoft
Sen Yang Microsoft
Jingren Zhou* Microsoft
Lidong Zhou Microsoft Research

Abstract

StreamScope (or StreamS) is a reliable distributed stream computation engine that has been deployed in shared 20,000-server production clusters at Microsoft. StreamS provides a continuous temporal stream model that allows users to express complex stream processing logic naturally and declaratively. StreamS supports business-critical streaming applications that can process tens of billions (or tens of terabytes) of input events per day continuously with complex logic involving tens of temporal joins, aggregations, and sophisticated user-defined functions, while maintaining tens of terabytes in-memory computation states on thousands of machines.

StreamS introduces two abstractions, rVertex and rStream, to manage the complexity in distributed stream computation systems. The abstractions allow efficient and flexible distributed execution and failure recovery, make it easy to reason about correctness even with failures, and facilitate the development, debugging, and deployment of complex multi-stage streaming applications.

1 Introduction

An emerging trend in big data processing is to extract timely insights from continuous big data streams with distributed computation running on a large cluster of machines. Examples of such data streams include those from sensors, mobile devices, and on-line social media such as Twitter and Facebook. Such stream computations process infinite sequences of input events and produce timely output events continuously. Events are often processed in multiple stages that are organized into a directed acyclic graph (DAG), where a vertex corresponds to the continuous and often stateful computation in a stage and an edge indicates an event stream flowing downstream from the producing vertex to the consuming vertex. In contrast to batch processing, also often modeled as a DAG [27], a defining characteristic of cloud-scale stream computation is its ability to process potentially *infinite* input events *continuously* with delays in seconds and minutes, rather than processing a static data set in hours and days. The continuous, transient, and latency-sensitive nature of stream computation makes it challenging to cope with failures and variations that are typical in a large-scale distributed system, and makes stream applications hard to develop, debug, and deploy.

This paper presents the design and implementation of StreamScope (or StreamS), a cloud-scale reliable stream computation engine that has been deployed in shared production clusters, each containing over 20,000 commodity servers. StreamS adopts a declarative language that supports a continuous stream computation model, extended with the ability to allow user-defined functions to customize stream computation at each step.

StreamS has been designed for business-critical stream applications desiring a strong guarantee that each event is processed exactly once despite server failures and message losses. Failure recovery in cloud-scale stream computation is particularly challenging because of two types of dependencies: the dependency between upstream and downstream vertices, and the dependency introduced by vertex computation states. An upstream vertex failure affects downstream vertices directly, while the recovery of a downstream vertex would depend on the output events from the upstream vertices. Failure recovery of a vertex would require rebuilding the state before the vertex can continue processing new events. StreamS therefore introduces two new abstractions, rVertex and rStream, to manage the complexity of cloud-scale stream computation by addressing the two types of dependencies through decoupling. rVertex models continuous computation on each vertex, introduces the notion of *snapshots* along the time dimension, and allows the computation to restart from a snapshot. rStream abstracts out the data and communication aspects of distributed stream computation, provides the illusion of reli-

[*] Now with Alibaba Group.

able and asynchronous communication channels, and decouples upstream and downstream vertices. Combined, rVertex and rStream offer well-defined semantics to replay computation and to rewind streams, as needed during failure recovery, thereby making it easy to develop different failure recovery strategies while ensuring correctness. The power of this abstraction also comes from the separation of its properties from its actual implementation that achieves those properties. That, for example, allows a different implementation of rStream specifically for development and debugging.

Our evaluation shows that StreamS can support complex production streaming applications deployed on thousands of machines, processing tens of billions of events per day with complex computation logic to deliver business-critical results continuously despite unexpected failures and planned maintenance, while at the same time demonstrating good scalability and capability of achieving 10-millisecond latencies on simple applications.

StreamS's key contributions are as follows. First, StreamS shows that a cloud-scale distributed fault-tolerant stream engine can support a continuous stream computation model without having to converting a stream computation unnaturally to a series of mini-batch jobs [42]. Second, StreamS introduces two new abstractions, rVertex and rStream, to simplify cloud-scale stream computation engines through separation of concerns, making it easy to understand and reason about correctness despite failures. The abstractions are also effective in addressing challenges on debugging and deployment of stream applications in StreamS. Support for debugging and deployment is critical in practice from our experiences, but has not received sufficient attention. Finally, StreamS is deployed in production and runs critical stream applications continuously on thousands of machines while coping well with failures and variations.

The rest of the paper is organized as follows. Section 2 describes StreamS's continuous stream model and declarative language. Section 3 defines StreamS's new abstractions: rVertex and rStream. Section 4 describes StreamS's design and implementation in detail, followed by a discussion of several design choices in Section 5. Engineering experiences are the topic of Section 6. Section 7 presents the evaluation results of StreamS in a production environment. We survey related work in Section 8 and conclude in Section 9.

2 Programming Model

In this section, we provide a high-level overview of the programming model, highlighting the key concepts including the data model and query language.

Continuous event streams. In StreamS, data is represented as event streams, each describing a potentially infinite collection of events that changes over time. Each event stream has a well-defined *schema*. In addition, each event has a time interval $[V_s, V_e)$, describing the start and end time for which the event is valid.

```
AlertWithUserID =
    SELECT Alert.Name AS Name, Process.UserID AS UserID
    FROM Process INNER JOIN Alert
    ON Process.ProcessID == Alert.ProcessID;

CountAlerts =
    SELECT UserID, COUNT(*) AS AlertCount
    FROM AlertWithUserID
    GROUP BY UserID
    WITH HOPPING(5s, 5s);
```

Figure 1: A simplified StreamS program.

Like other stream processing engines [9, 19], StreamS supports *Current Time Increments* (CTI) events that assert the completeness of event delivery up to start time V_s of the CTI event; that is, there will be no events with a timestamp lower than V_s in the stream after this CTI event. Stream operators rely on CTI events to determine the current processing time in order to make progress and to retire obsolete state information.

Declarative query language. StreamS provides a declarative language for users to program their applications without having to worry about distributed system details such as scalability or fault tolerance. Specifically, we extend the SCOPE [43] query language to support a full temporal relational algebra [15], extensible through user-defined functions, aggregators, and operators.

StreamS supports a comprehensive set of relational operators including projection, filters, grouping, and joins, adapted for temporal semantics. For example, a temporal inner join applies to events with overlapping time intervals only. Windowing is another key concept in stream processing. A window specification defines time windows and consequently defines a subset of events in a window, to which aggregations can be applied. StreamS supports several types of time-based windows, such as hopping, tumbling, and snapshot windows. For example, hopping windows are windows (of size S) that "jump" forward in time by a fixed size H: a new window of size S is created for every H units of time.

Example. Figure 1 shows an example program that performs continuous activity diagnosis on Process and Alert event streams. A StreamS program consists of a sequence of declarative queries operating on event streams. Process events record information about every process and its associated user, while Alert events record information about every alert, including which process generated the alert. The program first joins the two streams to attach user information to alerts, and then calculates for each user the number of alerts every 5 seconds using a hopping window.

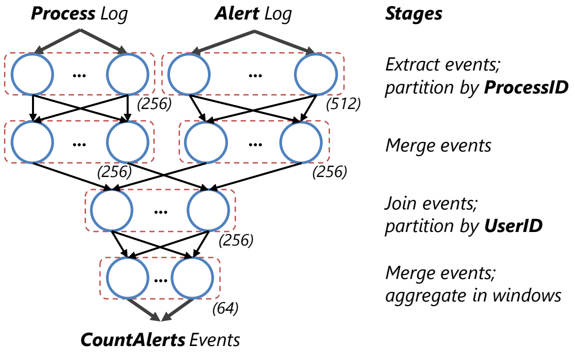


Figure 2: An execution DAG for the example in Figure 1.

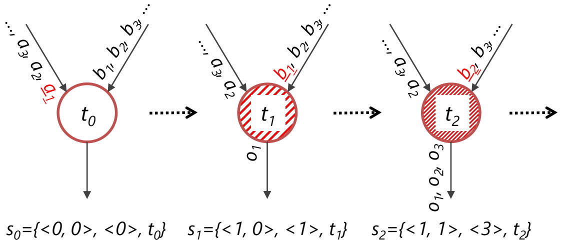


Figure 3: Vertex execution from snapshot s_0 to s_2.

3 STREAMS Abstractions

The execution of a STREAMS program can be modeled as a directed acyclic graph (DAG), where each *vertex* performs local computation on *input streams* from its in-edges and produces *output streams* as its out-edges. Each stream is modeled as an infinite sequence of events, each with a continuously incremented *sequence number*. Figure 2 shows an example DAG corresponding to Figure 1, where each stage of computation is partitioned into multiple vertices to execute in parallel. STREAMS determines the degree of parallelism for each stage (marked in parentheses) based on data rate and computation cost.

A vertex can maintain a local state. Its execution starts with its initial state and proceeds in steps. In each step, the vertex consumes the next events from its input streams, updates its state, and possibly produces events to its output streams. The execution of a vertex is tracked through a series of *snapshots*, where each snapshot is a triplet containing the current sequence numbers of its input streams, the current sequence numbers of its output streams, and its current state. Figure 3 illustrates the progression of a vertex execution from snapshot s_0, to s_1 (after processing a_1), and then to s_2 (after processing b_1). STREAMS introduces two abstractions rStream and rVertex to implement streams and vertices, respectively.

3.1 The rStream Abstraction

Rather than having vertices communicate directly through the network, STREAMS introduces an rStream abstraction to decouple upstream and downstream vertices with properties to facilitate failure recovery.

Conceptually, rStream reliably maintains a sequence of events with continuous and monotonically increasing sequence numbers, supporting multiple writers and readers. A writer issues `Write(seq,e)` to add event `e` with sequence number `seq`. rStream supports multiple writers mainly to allow two instances of the same vertex, which is useful when handling failures and stragglers via duplicated execution, as described in Section 4.

A reader can issue `Register(seq)` to indicate its interest in receiving events starting from sequence number `seq` and start reading from the stream using `ReadNext()`, which returns the next batch of events with their sequence numbers and advances the reading position accordingly. In the implementation, events can be pushed to a registered reader rather than pulled. With rStream each reader can proceed asynchronously from the same stream without synchronizing with other readers or writers. A reader can also rewind a stream by re-registering with an earlier sequence number (e.g., for failure recovery). rStream also supports `GarbageCollect(seq)` to indicate that all events less than sequence number `seq` will not be requested any more and therefore can be discarded. rStream maintains the following properties.

Uniqueness. There is a unique value associated with each sequence number. After the first write for each sequence number `seq` succeeds, any subsequent write that associates `seq` will be discarded.

Validity. If a `ReadNext()` returns an event `e` with sequence number `seq`, there must have been a `Write(seq,e)` that has returned successfully.

Reliability. If `write(seq,e)` succeeds, then, for any `ReadNext()` reaching position `seq`, eventually the read returns `(seq,e)`.

Uniqueness ensures consistency for each sequence number, Validity ensures correctness of the event value returned for each sequence number, while Reliability ensures that, all events written to the stream are always available to readers whenever requested. rStream could simply be implemented by a reliable pub/sub system backed by reliable and persistent store. But STREAMS adopts a more efficient implementation that avoids paying the latency cost of going to persistent and reliable store in the critical path, with the additional mechanism of reconstructing the requested events through recomputation [38, 42], as detailed in Section 4.

3.2 The rVertex Abstraction

The rVertex abstraction supports the following operations for a vertex. `Load(s)` starts an instance of the vertex at snapshot `s`. `Execute()` executes a step from the current snapshot. `GetSnapshot()` returns the current snapshot. A vertex can then be started with Load(s_0), where s_0 is the initial snapshot with an initial state and with all streams at starting positions. The vertex can then execute a series of `Execute()` operations, which read the input events, update the state, and produce output events. At any point, one can issue `GetSnapshot()` to retrieve and save the snapshot. When the vertex fails, it can be restarted with `Load(s)`, where `s` is a saved snapshot.

Determinism. For a vertex with its given input streams, running `Execute()` on the same snapshot will always cause the vertex to transition into the same new snapshot and produce the same output events.

Determinism ensures correctness when replaying an execution during failure recovery. It implies that (i) the order in which the execution takes the next event from multiple input streams is deterministic; we explain how this order determinism is enforced naturally in StreamS without introducing unnecessary delay in Section 4, and (ii) the execution of the processing logic is deterministic. Determinism greatly simplifies the reasoning of correctness in StreamS and makes streaming applications easier to develop and debug. In Section 5, we discuss how to mask non-determinism when needed.

3.3 Failure Recovery

The rStream abstraction decouples upstream and downstream vertices to allow individual vertices to recover from failures separately. When a vertex fails, we can simply restart its execution by calling `Load(s)` from a most recently saved snapshot `s` to continue executing. The rVertex abstraction ensures that execution after recovery is the same as the continuation of the original execution as if no failures occurred. The rStream abstraction ensures that the restarted vertex is able to (re-)read the input streams. Section 4 describes how rVertex and rStream are implemented and how different failure recovery strategies can achieve different tradeoffs.

4 Architecture and Implementation

StreamS is designed and implemented as a streaming extension of the SCOPE [43] batch-processing system. As a result, StreamS heavily leverages the architecture, compiler, optimizer, and job manager in SCOPE, adapted or re-designed to support stream processing at scale. This approach expedites the development of StreamS; the integration of batch and stream processing also offers significant benefits in practice, as elaborated in Section 6.

In StreamS, a user programs a stream application declaratively as described in Section 2. The program is compiled into a streaming DAG for distributed execution as shown in Figure 4. To generate such a DAG, the StreamS *compiler* performs the following steps: (1) the program is first converted into a *logical* plan (DAG) of StreamS runtime operators, which include temporal joins, window aggregates, and user-defined functions; (2) the StreamS *optimizer* then evaluates various plans, choosing the one with the lowest estimated cost based on available resource, data statistics such as the incoming rate, and an internal cost model; and (3) a *physical* plan (DAG) is finally created by mapping a logical vertex into an appropriate number of physical vertices for parallel execution and scaling, with code generated for each vertex to be deployed in a cluster and process input events continuously at runtime. We omit the details of these steps as they are similar to those for SCOPE [43].

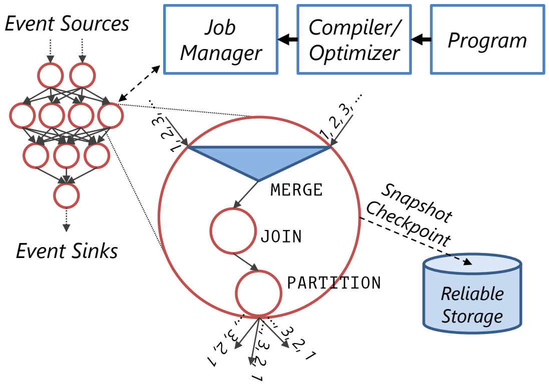


Figure 4: An overview of a StreamS program.

The entire execution is orchestrated by a streaming *job manager* that is responsible for: (1) scheduling vertices to and establishing channels (edges) in the DAG among different machines; (2) monitoring progress and tracking snapshots; (3) providing fault tolerance by detecting failures/stragglers and initiating recovery actions. Unlike a batch-oriented job manager that schedules vertices at different times on demand, a streaming job manager schedules all vertices in a DAG at the beginning of job execution. To provide fault tolerance and to cope with runtime dynamics, rVertex and rStream are used to implement vertices and channels in a streaming DAG, working in coordination with the job manager.

4.1 Implementing rVertex

The key to implementing rVertex is to ensure Determinism as defined in Section 3, which requires both *function determinism* and *input determinism*. In StreamS, all operators and user-defined functions must be deterministic. We also assume that the input streams for a job are deterministic, both in terms of order and event values. The only remaining input-related non-determinism is the ordering of events across multiple input streams. Because StreamS uses `CTI` events (Section 2) as markers, we insert a special `MERGE` operator at the beginning of a vertex that takes multiple input streams, which produces a deterministic order of events for subsequent processing in the vertex. It does so by waiting for the corresponding CTI events across input streams to show up, ordering them deterministically, and emitting them in that deterministic order. Because the processing logic of vertices tends to wait for the CTI events in the same way, this solution does not introduce additional noticeable delay.

STREAMS labels events in each stream with consecutive monotonically increasing sequence numbers. A vertex uses sequence numbers to track the last consumed/produced events from all streams. At each step of the execution, a vertex consumes the next event(s) of the input streams, invokes Execute(), which might change its internal state, and generates new events into the output streams, thereby reaching a new snapshot. GetSnapshot() returns such a snapshot, which can be implemented by pausing the execution after a step or some copy-on-write data structures so that a consistent snapshot can be retrieved while running uninterrupted. Load(c) starts a vertex and loads c as the current snapshot before resuming execution. To be able to resume execution from a snapshot, a vertex can periodically create a *checkpoint* and store it reliably and persistently.

4.2 Implementing rStream

The rStream abstraction provides reliable channels that allow receivers to read from any written position. One straightforward implementation is for producing vertices to write events persistently and reliably into the underlying Cosmos distributed file system. Those synchronous writes introduce significant latencies in the critical path of stream processing. STREAMS instead uses a hybrid scheme that moves those writes out of the critical path while providing the illusion of reliable channels: the events being written are first buffered in memory co-located with the producing vertex and can be transmitted directly to consuming vertices. The in-memory buffer is asynchronously flushed to Cosmos to survive server failures. Events that are only kept in memory might be lost on a failure, but can be recomputed when requested.

To be able to recompute lost events in case of failures, STREAMS tracks how each event is computed, similar to dependency tracking in TimeStream [38] or lineage in D-Streams [42]. In particular, during execution, the job manager tracks vertex snapshots (through GetSnapshot()), which it can use later to infer how to reproduce events in the output streams. A vertex decides for itself when to capture a snapshot, save it (e.g., checkpointing to a reliable persistent store), and report progress to the job manager. For example, in Figure 5, vertex v_4 sends two updates to the job manager. The first update reports a snapshot $s_1 = \{\langle 2,7\rangle,\langle 12\rangle,t_1\}$, which indicates that the vertex consumed up to event 2 in the first input stream and up to event 7 in the second, and produced output event 12, while at state t_1. The second update $s_2 = \{\langle 5,10\rangle,\langle 20\rangle,t_2\}$ reports that it has reached event 5 in the first input stream, event 10 in the second, while at state t_2. This tracking is completely transparent to users. Now if event 16 in the output stream needs to be recomputed, the job manager can simply scan the snapshots and find the highest output sequence number that is lower than 16, which is s_1 in this case. It then starts a new instance of the vertex, loads snapshot s_1, and continues executing until event 16 is produced. The execution of that new instance would require events 3-5 from the first input stream and events 8-10 in the second, which might trigger recomputation in upstream vertices if those events were no longer available. This process eventually terminates as the original input events are always assumed to be reliably persisted. Overall, such a design moves the flush to the reliable persistent store out of the critical path in normal execution, while at the same time reduces the number of events that need to be recomputed during failure recovery. While rStream is conceptually infinite, in a real implementation, garbage collection is necessary to remove obsolete events and related tracking information that are no longer needed for producing output events or for failure recovery, which we describe in Section 4.3.

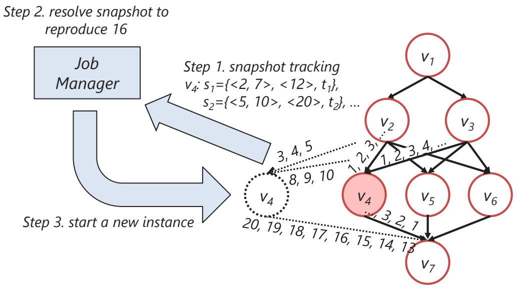

Step 2. resolve snapshot to reproduce 16
Job Manager
Step 1. snapshot tracking
v_4: s_1={<2, 7>, <12>, t_1},
s_2={<5, 10>, <20>, t_2}, ...
Step 3. start a new instance
v_1
v_2
v_3
v_4
v_4
v_5
v_6
v_7
3, 4, 5
8, 9, 10
1, 2, 3, ...
1, 2, 3, 4, ...
..., 3, 2, 1
20, 19, 18, 17, 16, 15, 14, 13

Figure 5: Snapshot tracking and recovery for rStream.

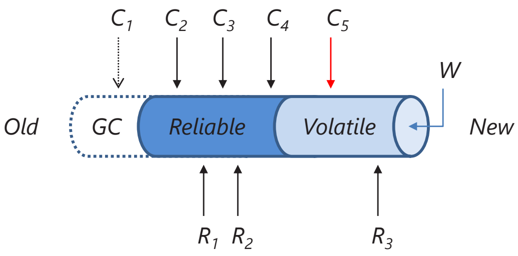


Figure 6: The STREAMS implementation of rStream.

Figure 6 illustrates rStream's implementation. In this example, there is one writer W (the upstream vertex) and three readers R_1, R_2, and R_3 (downstream vertices). The stream grows over time from left to right. The prefix of the stream (marked as GC) includes events that are

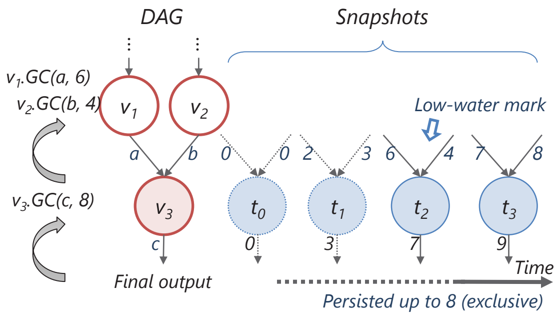


Figure 7: Garbage collection: a recursive view.

obsolete and can be garbage collected, followed by a sequence of events that have been reliably persisted. The tail of the stream (i.e., the most recent events) is *volatile* and could get lost when failures happen. Checkpoints can be created periodically (e.g., C_1, C_2, C_3, C_4, and C_5) for snapshots of the upstream vertex. When the volatile portion of the stream is lost due to failure, it can be recomputed from snapshot C_4. Events in the reliable portion can be served to R_1 and R_2 without recomputation. Replaying from a checkpoint in the reliable portion (e.g., C_4) fully recovers the transient portion; as a result, no further cascading recomputation is needed for recovery.

4.3 Garbage Collection

STREAMS persists checkpoints of snapshots, streams, and other tracking information reliably for failure recovery, and must determine when such information can be garbage collected. STREAMS maintains *low-water marks* for vertices and streams during the execution of a streaming application. For a stream, the low-water mark points to the lowest sequence number of the events that are needed; for a vertex, the low-water mark points to the lowest snapshot of the vertex that are needed.

For each vertex, snapshots are totally ordered by the vector of sequence numbers of its input and output streams. Because snapshots capture the linear progress of deterministic vertex execution, all sequence numbers only move forward. For example, consider a vertex with two input streams and one output stream, a snapshot s with a vector of sequence numbers at $(\langle 7,12\rangle,\langle 5\rangle)$ will be lower than a snapshot at $(\langle 7,20\rangle,\langle 8\rangle)$. There cannot be another snapshot at $(\langle 6,16\rangle,\langle 4\rangle)$.

Consider a vertex v with I as its set of input streams and O as its set of output streams. Vertex v maintains a low-water mark sequence number lm_o for each output stream $o \in O$, initialized to 0. Vertex v implements $\texttt{GC}(o,m)$ to perform garbage collection, indicating that any sequence number lower than m will no longer be requested by the downstream vertex consuming output stream o. For simplicity, we assume that each stream is consumed by a single downstream vertex, but it is straightforward to support the general case, where a stream is shared among multiple downstream vertices.

1. If $m \leq lm_o$, return; // no further GC needed.

2. Set lm_o to m. Let s be the highest checkpointed snapshot s satisfying the condition that the sequence number for output stream o in s is no higher than lm_o for every $o \in O$. Discard any snapshot lower than s.

3. For each input stream $i \in I$, let v_i be the upstream vertex producing input stream i and let s_i be the sequence number corresponding to input stream i in s, call $v_i.\texttt{GarbageCollect}\ (s_i)$ to discard events lower than s_i in input stream i. Recursively call $v_i.\texttt{GC}(i,s_i)$.

Intuitively, $\texttt{GC}(o,m)$ figures out which information is no longer needed if the downstream vertex (connected to the output stream o) will not request any events with a sequence number lower than m. It is called when the final output events are persisted or consumed, or when any output events in a stream is persisted reliably. Figure 7 shows an example of low-water marks. Although the algorithm is specified recursively, it can be implemented efficiently through a reverse topological order traversal.

4.4 Failure Recovery Strategies

STREAMS must recover from failures to keep streaming applications running. The rVertex and rStream abstractions decouple downstream vertices in a DAG from their upstream counterparts, making it easier to reason about and deal with runtime failures. In addition, they abstract away underlying implementation details, and allow them to share a common mechanism for fault tolerance.

Different failure recovery strategies can be developed; the choice can be decided by a combination of factors: normal-case cost (in terms of resources required), normal-case overhead (in terms of latency), recovery cost (in terms of resources required for recovery), and recovery time. We highlight three strategies that represent different tradeoffs that are appropriate in different scenarios. With rStream and rVertex, each vertex can recover from failures independently. As a result, those strategies can be applied at the vertex granularity and even different vertices in the same job could potentially use different strategies due to their different characteristics.

Checkpoint-based recovery. In this strategy, a vertex checkpoints its snapshot periodically into a reliable persistent store. When the vertex fails, it will load the most recent checkpoint and resume execution. A straightforward implementation of checkpointing introduces overhead in normal execution that is not ideal for vertices that maintain a large internal state. Advanced checkpointing techniques [33, 34] often require specific data structures, which introduces complexity and overhead.

Replay-based recovery. Quite often stream computation is either stateless or has a *short-term memory* due to its use of window operators; that is, its current internal state depends only on the events in the most recent window of a certain duration (say the last 5 minutes). In those cases, a vertex can get away with not explicitly checkpointing state, and instead reloading that window of events to rebuild state from an initial one. While this is a special case, it is common enough to be useful. Leveraging this property, STREAMS can simply track the sequence numbers of the input/output streams without having to store the local states of a vertex. This strategy might need to reload a possibly large window of input events during recovery, but it avoids the upfront cost of checkpointing in the normal case.

This strategy has a subtle implication on garbage collection. Instead of loading a state in a snapshot, a vertex must recover it from earlier events in the input streams. Those events must be retained along with the snapshot.

Replication-based recovery. Yet another strategy is to have multiple instances of the same vertex run at the same time: they can be connected to the same input streams and output streams. Our rStream implementation allows multiple readers and writers, deduplicating automatically based on sequence numbers. The Determinism property of rVertex also makes replication a viable approach because those instances will behave consistently. With replication, a vertex can have instances take checkpoints in turn without affecting latency observed by readers because other instances are running at a normal pace. When one instance fails, it can also get the current snapshot from another instance directly to speed up recovery. All those benefits come at the cost of having multiple instances running at the same time.

5 Discussion

STREAMS makes different choices from existing distributed stream processing engines on stream model, non-determinism, and out-of-order event processing.

Mini-batch stream processing with RDD. Instead of supporting a continuous stream model, D-Streams [42] models a stream computation as a series of mini-batch computations in small time intervals and leverages immutable RDDs [41] for failure recovery. Modeling a stream computation as a series of mini-batches could be cumbersome because many stream operators, such as windowing, joins, and aggregations, maintain states to process event streams efficiently. Breaking such operations into separate mini-batch computation requires rebuilding computation state from the previous batches before processing new events in the current batch. A good example is the inner join in Figure 1. The join operator needs to track all the events that might generate matching results for the current or future batches in efficient data structures and can only retire them on CTI events. Regardless of how mini-batches are generated, such a join state is potentially big for complex join types and needs to be rebuilt in each batch or passed on between consecutive mini-batches. Furthermore, D-Streams unnecessarily couples low latency and fault tolerance: a mini-batch defines the granularity at which vertex computation is triggered and therefore dictates latency, while an immutable RDD, mainly for failure recovery, is created for each mini-batch. The low-latency requirements demand small batch sizes even though there is no need to enable failure recovery at that granularity.

Non-determinism. Determinism is required in rVertex for correctness and also makes debugging easier. Non-determinism could introduce inconsistency when a vertex re-executes during failure recovery. Non-determinism might cause re-execution to deviate from the initial execution and lead to a situation where downstream vertices use two inconsistent versions of the output event streams from this vertex. STREAMS can be extended to support non-determinism, but at a cost.

One way to avoid inconsistency due to non-determinism is to make sure that any output events produced by a vertex do not need to be recomputed. This can be achieved, for example, by checkpointing the snapshot to a reliable and persistent store before making the output events visible to downstream vertices. This is in essence the choice that MillWheel [7] makes in its design. This proposal introduces significant overhead because the expensive checkpointing is on the critical path. An alternative approach is to log non-deterministic decisions during execution for faithful replay [10, 22, 23, 32, 35]. Logging non-deterministic decisions is often less costly than checkpointing snapshots, but this approach requires that all sources of non-determinism be identified, appropriately logged, and replayed. STREAMS does not support such mechanisms in the current implementation.

Out-of-order event ordering. Events could arrive in an order that does not correspond to their application timestamps, for example, when the events come from multiple sources. To allow out-of-order event processing, systems such as Storm [3] and MillWheel [7] assign a unique but unordered ID to each event. A downstream vertex sends ACKs with those IDs to an upstream vertex to track progress and handle failures. STREAMS decouples the logical order of events from their physical delivery and consumption. It borrows the idea of CTI events as discussed in Section 2 from stream databases to achieve out-of-order event processing at the language and operator level. At the system level, STREAMS assigns unique and ordered sequence numbers to events, making it easy to track progress and handle failures, while avoiding explicit ACKs that could incur performance overhead.

6 Production Experiences

STREAMS has been deployed in production. This section highlights our experiences with developing STREAMS and with supporting the life cycle of streaming applications, from development, debugging, to deployment.

From batch to streaming. STREAMS has been developed as an extension to an existing large-scale batch processing system and benefited greatly from reusing the existing components, such as the compiler, optimizer, and DAG/job manager, with adaptation and changes to support streaming. For example, the compiler is extended to handle streaming operators and the optimizer has a revised cost function to evaluate streaming plans.

Quite a few streaming applications were migrated from recurring batch jobs to achieve better efficiency and low latency. STREAMS provides supports for such migration in the compiler and allows the use of a batch version to validate the results of a streaming counterpart.

Scaling and robustness to fluctuation. STREAMS creates a physical plan to handle scaling based on estimated peak input rates and operator costs, ensuring that a sufficient number of vertices in each stage execute in parallel to handle the peak load. We find STREAMS's design robust to fluctuations caused by load spikes or server failures thanks to the decoupling between vertices using rStream. When one vertex falls behind temporarily, the input events to the vertex are queued in essentially an infinite buffer in the underlying distributed storage system. The queuing also allows effective event batching to allow the vertex to catch up quickly. When the peak load increases over time, STREAMS provides the support to move to a new configuration with increased degrees of paralleliem, without any interruption. This is done by initiating new vertices with derived states from checkpoints in the current job, and retires the corresponding vertices when the new ones catch up. The flexibility of rStream makes it easy to support such transitions. We decided not to support dynamic reconfiguration [38] as the additional complexity was not justified.

Distributed streaming made easy. In STREAMS, the declarative programming language and the stream data model makes it easy to program a streaming application without worrying about the distributed system details. STREAMS extends the simplicity to development and debugging via a different instantiation of rStream and a different scheduling policy in the job manager. Specifically, STREAMS introduces an *off-line* mode, where finite datasets, usually persistently stored, can be read to simulate on-line event streams, through a special instantiation of rStream. The job manager also uses a special *off-line* mode that favors ease of debugging over latency by executing one vertex at a time, instead of running all vertices concurrently, thereby significantly reducing the required resources. The off-line mode is completely *transparent* to the user code, which behaves the same way as in the on-line version except for latency.

Traveling back in time. A streaming application typically progresses forward in time, but we have encountered cases where traveling back in time is needed. For example, a user might request to re-examine a segment of execution in the past in response to an audit request. As a result of the investigation, the user needs to apply adjustments to the past results because the learning algorithm used is imperfect and needs correction in this particular case. To improve the algorithm, the user further conducts experiments with new algorithms and compares them with the current one. To handle such requirements, we maintain all past checkpoints and input channels in a global *repository* that implements a retention policy. Our rVertex and rStream abstractions support time travel to the past as is the case for failure recovery.

Continuous operation during system maintenance. Cluster-wide maintenance and updates, e.g., to apply patches, occur regularly in data centers. For batch jobs, the maintenance can be done systematically by not assigning new tasks to the ones to be patched and waiting for the existing tasks to finish on those machines. This is unfortunately not sufficient for streaming applications as they run continuously. Instead, STREAMS leverages duplicate execution (as used to handle stragglers) to minimize the effect: after receiving a notification that certain machines are scheduled for maintenance, the job manager replicates the execution of each affected vertex and schedules another instance in a different safe machine. Once the new instance catches up in terms of events processing, the job manager can safely kill the affected vertex and allow maintenance to proceed.

Straggler handling. Stragglers are vertices that make progress at a slower rate than other vertices in the same stage. Preventing and mitigating stragglers is particularly important for streaming applications where stragglers can have more severe and long-lasting performance impact. First, STREAMS continuously monitors machine health and only considers healthy machines for running streaming vertices. This significantly reduces the likelihood of introducing stragglers at runtime. Second, for each vertex, STREAMS tracks its resulting CTI events, normalized by the number of its input events, to estimate roughly its progress. If one vertex has a processing speed that is significantly slower than the others in the same stage that execute the same query logic, a duplicate copy is started from the most recent checkpoint. They execute in parallel until either one catches up with the others, at which point the slower one is killed.

A streaming application might encounter anomalies during its execution; for example, when hitting unexpected input events. We have encountered cases where

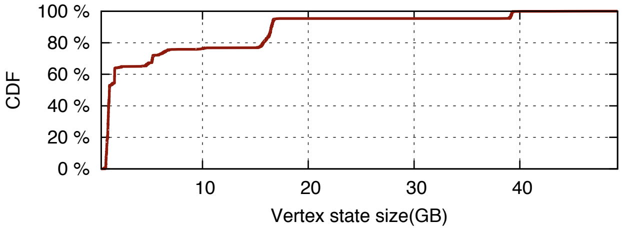


Figure 8: Distribution of vertex state sizes.

certain rare input events take a lot of time to process. The vertex hitting such an event is often considered a straggler, but such a straggler cannot be fixed by duplicate execution as the computation is always expensive. We extend STREAMS with an *alert* mechanism to supply users with various alerts, including event processing speeds and on-line statistics, and provide a flexible mechanism that allows users to specify a *filter* to weed out such events to keep the streaming application running smoothly (before a new solution is ready to be deployed). To ensure determinism, before a filter is applied, STREAMS creates a checkpoint for the vertex, flushes the volatile part of the streams, and records the filter.

7 Evaluation

STREAMS has been deployed since late 2014 in shared 20,000-server production clusters, running concurrently a few hundred thousand jobs daily, including a variety of batch, interactive, machine-learning, and streaming applications. Our evaluation starts with an in-depth study of a large business-critical production streaming application. Next, we perform extensive experiments using three simple streaming applications to demonstrate the scalability and performance with STREAMS, as well as the tradeoff between latency and throughput. Finally, we evaluate different failure recovery strategies. All the experiments are carried out in our shared production environment to perform real and practical evaluation.

7.1 A Production Streaming Application

For our evaluation, we study a production streaming application that supports a core on-line advertisement service. The application detects fraud clicks of on-line transactions in near-real time and refunds affected customers accordingly. Because the application is related to accounting, strong guarantees are needed. Latency is important for the application because lower latency allows customers to adjust their selling strategies more quickly, leading to higher revenues for the service. The previous implementation as a batch job introduced a latency of around 6 hours: it had to wait for the data to accumulate (every 3 hours) and to rebuild the state every time it ran.

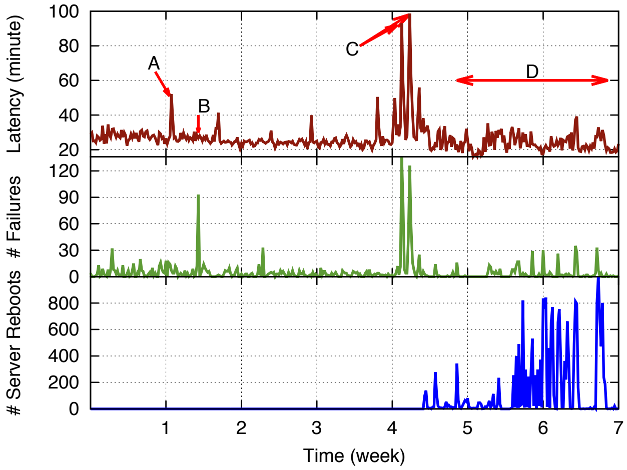


Figure 9: Application performance, failures, and server reboots over a 7-week period.

The application has a complex processing logic that involves a total of 48 stages, containing 18 joins of 5 different types (specifically, left semi, left anti semi, left outer, inner, and clip [9]). During the period of evaluation, the application executes on 3,220 vertices, processing tens of billions of input events (around 9.5 TB in size) and resulting in around 180 TB of I/O per day. Figure 8 shows the distribution of in-memory vertex state sizes in the application. There are about 25% "heavy" vertices that maintain a huge in-memory state because the application extracts millions of features from raw events and maintains a large number of complicated statistics in memory before feeding them into a sophisticated on-line prediction engine for fraud detection. The aggregated in-memory state of all the vertices is around 21.3 TB.

7.2 STREAMS in Production

In a shared production environment, failures, variations, and system maintenance are the norm. We cover several key aspects of the production streaming application, including performance, failures, variations, and stragglers.

Performance and failure impact. Figure 9 shows the end-to-end latency of this application over a 7-week period (top figure), along with the number of server failures (middle figure), and the number of servers brought down for planned maintenance (bottom figure), all aligned on time. We observed random server failures from time to time, impacting the application latency in various ways. We also experienced a major planned system maintenance that systematically rebooted machines.

We highlight four interesting periods, labeled *A*, *B*, *C*, and *D*. In case *A*, although the number of failures was not

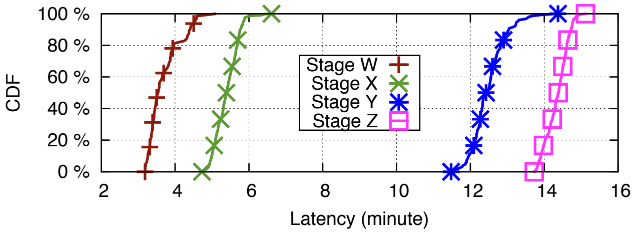
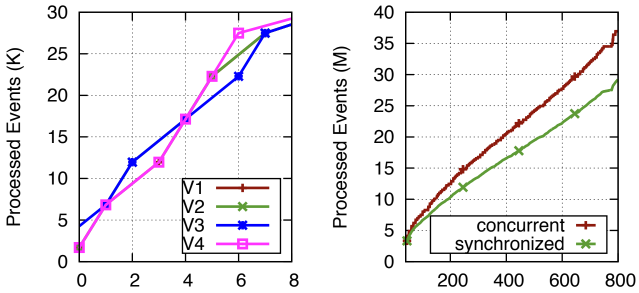


Figure 10: Distribution of vertex latency in four representative stages.



(a) Processing speed variations

(b) Synchronized v.s. concurrent channels

Figure 11: Benefits of concurrent channels.

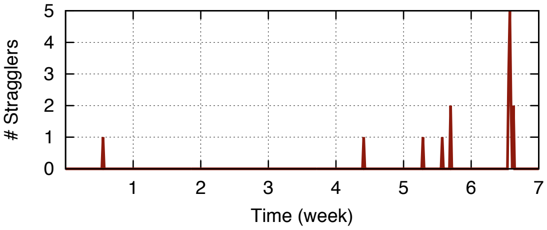


Figure 12: Stragglers in production.

high, some of the failed vertices held a relatively large in-memory state and took a long time to recover, thereby leading to a significant latency spike. In case *B*, however, a majority of failures occurred to vertices with a relatively small state. The recovery was therefore fast and its latency impact was hardly visible. Case *C* corresponds to a "live site" triggered by an *unplanned* mis-configuration that caused a significant number of machines to reboot. The issue led to a significant latency spike, but the application survived this massive failure. In Case *D*, a *planned* system maintenance rebooted machines in batches to apply OS patches and software upgrades. The entire application had to be migrated to run on a different set of machines. STREAMS used duplicate execution for each affected vertex to migrate gracefully.

The average end-to-end latency for the application is around 20 minutes. The original timestamps of the input events are included in the final output events and are used to compute end-to-end latency. The input delay, which is the interval between when events are generated/timestamped and when they appear in the input streams of the application, is also included. Window aggregations semantically introduce delay in latency and are the dominant factor in the overall latency for this application.

Variations. Performance variations are common in distributed computation, even among vertices in the same stage. Figure 10 shows the latency distribution of all the vertices in four representative stages, respectively. Stage *W* is responsible for extracting input events from raw event logs: the latency observed at that stage is mostly due to input delay. The variations observed in that stage are also consistently observed in later stages. Stage *X* contains window aggregations, which intrinsically introduce delays that are comparable to those of the window sizes to the downstream stage *Y*. Stage *Z* represents the application's final computation stage. Variations are the result of various factors: load fluctuation and interference on servers, or changing data characteristics and their impact on computation complexity and efficiency.

Concurrent channels. Interestingly, noticeable performance variations are observed on vertices that process the same data from the same upstream vertex. We examine a vertex whose output events are broadcast to 150 downstream vertices in the application we study. Figure 11(a) shows a detailed 8-minute view of processing speed variations on four selected vertices. A difference of several thousand events in processing speeds shows up from time to time, mainly due to computation variations in individual vertices: one vertex might outperform others for a period of time and then lag behind in the next. This observation argues against a naive *synchronized* design (e.g., using TCP directly) that forces all downstream vertices to proceed in lock steps, causing the slowest vertex to dictate in each step. STREAMS employs a *concurrent* design, allowing individual downstream vertices to advance at different speeds. Figure 11(b) compares the projected progress of such a synchronized design with the actual execution that uses a concurrent channel. The performance of using concurrent channels noticeably outperforms that of using synchronized ones.

Stragglers. Stragglers do appear in production even with mechanisms to prevent them. A straggler cannot recover by itself, unlike performance variation, and actions such as duplicate execution might be needed to resolve it. Figure 12 shows the number of stragglers we detected and successfully recovered during the 7-week period. We are conservative in classifying a vertex as a straggler because we observe that in most cases a vertex that falls behind temporarily can catch up by processing at large batches. The detected ones are those with persistent issues.

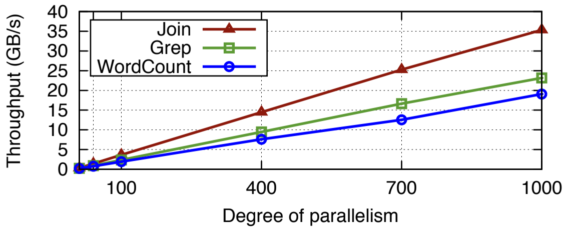


Figure 13: Scalability.

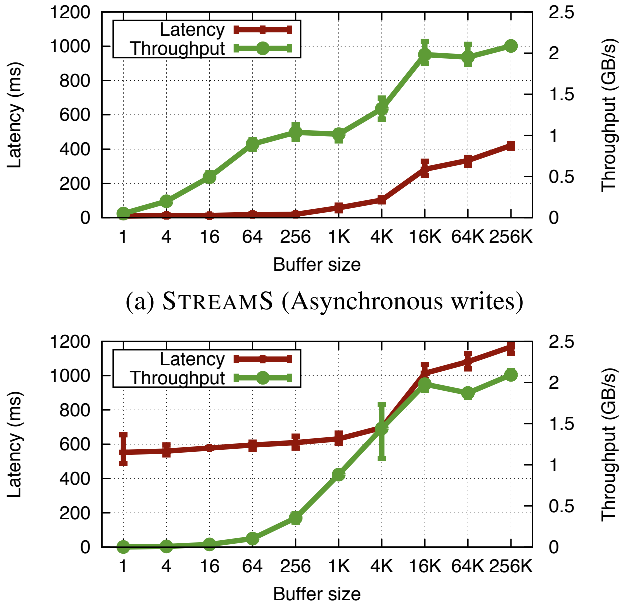
7.3 Scalability

To evaluate scalability and study performance tradeoff, we run three simple streaming applications in production. (1) **Grep** scans input events (strings) for a matching pattern; (2) **WordCount** counts the number of words in an input stream over 1-minute hopping windows, and (3) **Join** joins two input streams to return matching pairs. Each event in the first input stream has a 2-min window $[t, t+2]$, while a matching event with the same join key appears in the second stream in a 1-min window $[t+0.5, t+1.5]$, so that each event appears in the join result, allowing the application to produce a steady output stream. In all cases, each event is 100 bytes.

To evaluate scalability, we run each application with different numbers of vertices (degrees of parallelism) up to 1,000. We constrain each vertex to use one CPU core and limit I/O bandwidth to avoid significantly impacting production activities. Figure 13 reports the maximum throughput that STREAMS can sustain under a 1-second latency bound for each application with different numbers of vertices. STREAMS scales linearly to 1000 vertices, achieving a throughput of up to 35 GB/s. We also repeat the same experiments on a small dedicated 20-machine test cluster without any vertex resource constraint. STREAMS is able to saturate the network and the maximum throughput is bounded by network bandwidth.

7.4 Tradeoff: Latency vs. Throughput

We further study the tradeoff between latency and throughput by varying event buffer size using Grep with 100 vertices. We choose Grep because it has no window or join constructs that could introduce application-level delays. We repeat each experiment 3 times and report the average, minimum, and maximum values. As shown in Figure 14(a), STREAMS achieves a latency around 10 msec using a small buffer size at the cost of lower throughput. As the buffer size grows, the throughput improves but the latency also increases. STREAMS achieves a stable maximum throughout when buffering every 16K events, where the latency is around 280 msec. Our default production setting triggers computation either when accumulated events fill a 2MB buffer or every 500 msec, but it is configurable for each application.



(a) STREAMS (Asynchronous writes)

(b) Synchronous writes

Figure 14: Latency and throughput tradeoff using Grep with 100 vertices.

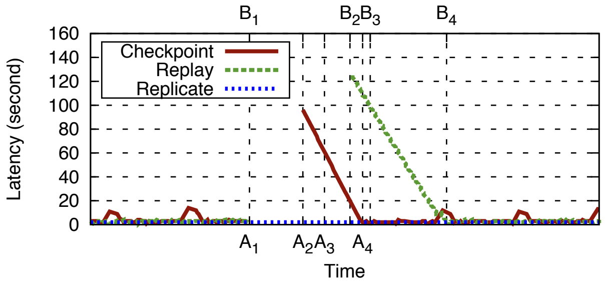


Figure 15: Comparing failure recovery strategies.

In STREAMS, events are first buffered in memory before *asynchronously* flushed to a reliable persistent store. To compare, we repeat the same Grep experiment by *synchronously* storing every event persistently, as done in MillWheel [7]. Figure 14(b) shows a similar tradeoff. However, its latency is always worse than that of STREAMS, while its throughput is worse when the buffer size is smaller than 1KB and is comparable otherwise.

7.5 Failure Recovery Strategies

We compare three failure recovery strategies using Join in Figure 15. The experiment is conducted in a production environment, where we inject a vertex failure manually, apply different recovery strategies, and observe the latency impact during failure and recovery. We align the time lines of the executions for ease of comparison.

The replication-based strategy has no impact on the latency because it always has at least two instances of the same vertex running. For the checkpoint-based strategy, each checkpointing introduces a small latency spike. After a failure, a new instance of the failed vertex reloads the latest checkpoint and continues executing. From A_1 to A_2, the checkpointed snapshot is reloaded. From A_2 to A_3, the vertex re-produces events that were already generated before failure. Those are discarded. After A_3, the vertex starts to produce new output events. The latency is high at this point because input events have been buffered during failure/recovery. The vertex catches up at A_4. Replay-based recovery does not have checkpointing overhead. The last snapshot is reconstructed by replaying the input events, which corresponds to the period between B_1 to B_2. There is a longer delay in replay-based recovery because the state in checkpoint is more condensed than the input events (a common case). Once the state is reconstructed, it follows the same steps as in the checkpoint-based recovery: it reproduces some duplicate output from B_2 to B_3 and then catches up at B_4. The actual shape of the curves depends on many factors, such as the sizes of the states, the number of events that must be replayed, the replay speed, and the catch-up speed.

As a rule of thumb, the checkpoint-based strategy is preferable if the checkpointing cost is low. The replay-based strategy is favored if the checkpointing cost is high, but the replay cost is comparable to that of recovery from a checkpoint. In our production application, 25% of the vertices use replay-based recovery (manually configured) to avoid the normal-case latency penalty, while the remaining use checkpoint-based recovery for fast fail-over. Replication is used for duplicate execution to handle stragglers or to enable migration.

8 Related Work

The key concepts in STREAMS, such as declarative SQL-like language, temporal relational algebra, compilation and optimization to streaming DAG, and scheduling, have been inherited from the design of stream database systems [6, 13, 20, 5, 19], which extensively studied stream semantics [29, 18, 15, 28] and distributed processing [11, 25, 17, 14, 26]. STREAMS's novelty is in the new rStream and rVertex abstractions designed for high scalability and fault tolerance through decoupling, representing a different and increasingly important design point that favors scalability at the expense of somewhat loosened latency requirements. MapReduce Online [21], S4 [37], and Storm [3, 31] extend a DAG model in batch processing systems like Hadoop [1] to streaming.

STREAMS is designed to achieve the exactly-once semantics despite failures; such strong consistency is required by many production streaming applications and makes it easy to reason about correctness. Other systems such as Trident [4] (over Storm), MillWheel [7, 8], TimeStream [38], D-Streams [42], and Samza [2] also embrace strong consistency, but make different design choices. The abstractions in STREAMS separate the key requirements of fault tolerant streaming processing from the different approaches in satisfying those requirements. For example, state management and tracking in Trident, MillWheel, and Samza can be considered a way to realize rVertex. TimeStream's dependency tracking and D-Streams' lineage tracking can be used to implement rStream with on-demand recomputation, while Kafka [30]-based channel implementation in Samza implements rStream with all events persisted reliably. STREAMS's rStream implementation moves the cost of reliable persistence out of the critical path (unlike MillWheel [7] and Samza [2]), while keeping the probability of on-demand recomputation low and avoiding cascading recovery in practice. Other noteworthy technical differences, such as continuous vs. mini-batch models, non-determinism, and out-of-order event processing, have been discussed in Section 5.

Several other systems focus on other design dimensions. For example, Photon [12] and JetStream [39] address the geo-distribution aspect of streaming to achieve consistency and efficiency over a wide area network. Naiad [36] and Flink [16] handle dataflows with cycles for incremental computation. SEEP [24] and ChronoStream [40] address the resource elasticity for streaming by dynamically adjusting the degree of parallelism. Heron [31] improves on Storm to run in shared production cluster efficiently and introduces backpressure.

9 Conclusion

STREAMS takes a principled approach to distributed fault-tolerant cloud scale stream computation with new abstractions rVertex and rStream. Its implementation and deployment in production not only provide the insights that validate the design choices, but also offer valuable engineering experiences that are key to the success of such a cloud scale stream computation system.

10 Acknowledgments

We would like to thank the anomymous reviewers, as well as our shepherd Hari Balakrishnan, for their valuable comments and suggestions. We are grateful to Michael Levin, Andrew Baumann, Geoff Voelker, and Jay Lorch for providing insightful feedback on our early draft. We would also like to thank Microsoft Big Data team members for their support and collaboration.

References

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Apache Samza. http://samza.apache.org/.

[3] Apache Storm. http://storm.incubator.apache.org/.

[4] Trident. https://storm.apache.org/documentation/trident-tutorial.html.

[5] ABADI, D. J., AHMAD, Y., BALAZINSKA, M., ÇETINTEMEL, U., CHERNIACK, M., HWANG, J., LINDNER, W., MASKEY, A., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. B. The design of the Borealis stream processing engine. In *CIDR* (2005), pp. 277–289.

[6] ABADI, D. J., CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. B. Aurora: A new model and architecture for data stream management. *VLDB J. 12*, 2 (2003), 120–139.

[7] AKIDAU, T., BALIKOV, A., BEKIROGLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., AND WHITTLE, S. MillWheel: Fault-tolerant stream processing at Internet scale. *PVLDB 6*, 11 (2013), 1033–1044.

[8] AKIDAU, T., BRADSHAW, R., CHAMBERS, C., CHERNYAK, S., FERNÁNDEZ-MOCTEZUMA, R. J., LAX, R., MCVEETY, S., MILLS, D., PERRY, F., SCHMIDT, E., ET AL. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1792–1803.

[9] ALI, M. H., CHANDRAMOULI, B., GOLDSTEIN, J., AND SCHINDLAUER, R. The extensibility framework in Microsoft StreamInsight. In *ICDE* (2011), pp. 1242–1253.

[10] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009* (2009), pp. 193–206.

[11] AMINI, L., ANDRADE, H., BHAGWAN, R., ESKESEN, F., KING, R., SELO, P., PARK, Y., AND VENKATRAMANI, C. SPC: A distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms* (2006), ACM, pp. 27–37.

[12] ANANTHANARAYANAN, R., BASKER, V., DAS, S., GUPTA, A., JIANG, H., QIU, T., REZNICHENKO, A., RYABKOV, D., SINGH, M., AND VENKATARAMAN, S. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), pp. 577–588.

[13] ARASU, A., BABCOCK, B., BABU, S., DATAR, M., ITO, K., NISHIZAWA, I., ROSENSTEIN, J., AND WIDOM, J. STREAM: The Stanford stream data manager. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003* (2003), p. 665.

[14] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S., AND STONEBRAKER, M. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *ACM SIGMOD Conf.* (Baltimore, MD, June 2005).

[15] BARGA, R. S., GOLDSTEIN, J., ALI, M. H., AND HONG, M. Consistent streaming through time: A vision for event stream processing. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings* (2007), pp. 363–374.

[16] CARBONE, P., FÓRA, G., EWEN, S., HARIDI, S., AND TZOUMAS, K. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603* (2015).

[17] CETINTEMEL, U. The Aurora and Medusa projects. *Data Engineering 51*, 3 (2003).

[18] CHAKRAVARTHY, S., KRISHNAPRASAD, V., ANWAR, E., AND KIM, S. Composite events for active databases: Semantics, contexts and detection. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile* (1994), pp. 606–617.

[19] CHANDRAMOULI, B., GOLDSTEIN, J., BARNETT, M., DELINE, R., PLATT, J. C., TERWILLIGER, J. F., AND WERNSING, J. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB 8*, 4 (2014), 401–412.

[20] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., and Shah, M. A. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), ACM, pp. 668–668.

[21] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. Mapreduce online. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA* (2010), pp. 313–328.

[22] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay.

[23] Dunlap, G. W., Lucchetti, D. G., Fetterman, M. A., and Chen, P. M. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008, Seattle, WA, USA, March 5-7, 2008* (2008), pp. 121–130.

[24] Fernandez, R. C., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), pp. 725–736.

[25] Franklin, M. J., Jeffery, S. R., Krishnamurthy, S., Reiss, F., Rizvi, S., Wu, E., Cooper, O., Edakkunni, A., and Hong, W. Design considerations for high fan-in systems: The HiFi approach. In *CIDR* (2005), pp. 290–304.

[26] Hwang, J.-H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., and Zdonik, S. High-Availability Algorithms for Distributed Stream Processing. In *The 21st International Conference on Data Engineering (ICDE 2005)* (Tokyo, Japan, April 2005).

[27] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007* (2007), pp. 59–72.

[28] Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H., Çetintemel, U., Cherniack, M., Tibbetts, R., and Zdonik, S. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1379–1390.

[29] Jensen, C. S., and Snodgrass, R. T. Temporal specialization. In *Proceedings of the Eighth International Conference on Data Engineering, February 3-7, 1992, Tempe, Arizona* (1992), pp. 594–603.

[30] Kreps, J., Narkhede, N., and Rao, J. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece* (2011).

[31] Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J. M., Ramasamy, K., and Taneja, S. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 239–250.

[32] Laadan, O., Viennot, N., and Nieh, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *SIGMETRICS 2010, Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, New York, USA, 14-18 June 2010* (2010), pp. 155–166.

[33] Li, K., Naughton, J. F., and Plank, J. S. *Real-time, concurrent checkpoint for parallel programs*, vol. 25. ACM, 1990.

[34] Li, K., Naughton, J. F., and Plank, J. S. Low-latency, concurrent checkpointing for parallel programs. *Parallel and Distributed Systems, IEEE Transactions on 5*, 8 (1994), 874–879.

[35] Montesinos, P., Hicks, M., King, S. T., and Torrellas, J. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009* (2009), pp. 73–84.

[36] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In *ACM SIGOPS*

24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013 (2013), pp. 439–455.

[37] NEUMEYER, L., ROBBINS, B., NAIR, A., AND KESARI, A. S4: Distributed stream computing platform. In *ICDMW 2010, The 10th IEEE International Conference on Data Mining Workshops, Sydney, Australia, 13 December 2010* (2010), pp. 170–177.

[38] QIAN, Z., HE, Y., SU, C., WU, Z., ZHU, H., ZHANG, T., ZHOU, L., YU, Y., AND ZHANG, Z. TimeStream: Reliable stream computation in the cloud. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013* (2013), pp. 1–14.

[39] RABKIN, A., ARYE, M., SEN, S., PAI, V. S., AND FREEDMAN, M. J. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), pp. 275–288.

[40] WU, Y., AND TAN, K.-L. ChronoStream: Elastic stateful stream computation in the cloud. In *Data Engineering (ICDE), 2015 IEEE 31th International Conference on* (2015), IEEE.

[41] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012* (2012), pp. 15–28.

[42] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized Streams: Fault-tolerant streaming computation at scale. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 423–438.

[43] ZHOU, J., BRUNO, N., WU, M.-C., LARSON, P.-Å., CHAIKEN, R., AND SHAKIB, D. SCOPE: Parallel databases meet MapReduce. *VLDB J. 21*, 5 (2012), 611–636.

Social Hash: an Assignment Framework for Optimizing Distributed Systems Operations on Social Networks

Alon Shalita[†], Brian Karrer[†], Igor Kabiljo[†], Arun Sharma[†], Alessandro Presta[†], Aaron Adcock[†], Herald Kllapi[*], and Michael Stumm[§]

[†]Facebook {alon,briankarrer,ikabiljo,asharma,alessandro,aadcock}@fb.com
[*]University of Athens herald@di.uoa.gr
[§]University of Toronto stumm@eecg.toronto.edu

Abstract

How objects are assigned to components in a distributed system can have a significant impact on performance and resource usage. *Social Hash* is a framework for producing, serving, and maintaining assignments of objects to components so as to optimize the operations of large social networks, such as Facebook's Social Graph. The framework uses a two-level scheme to decouple compute-intensive optimization from relatively low-overhead dynamic adaptation. The optimization at the first level occurs on a slow timescale, and in our applications is based on graph partitioning in order to leverage the structure of the social network. The dynamic adaptation at the second level takes place frequently to adapt to changes in access patterns and infrastructure, with the goal of balancing component loads.

We demonstrate the effectiveness of Social Hash with two real applications. The first assigns HTTP requests to individual compute clusters with the goal of minimizing the (memory-based) cache miss rate; Social Hash decreased the cache miss rate of production workloads by 25%. The second application assigns data records to storage subsystems with the goal of minimizing the number of storage subsystems that need to be accessed on multiget fetch requests; Social Hash cut the average response time in half on production workloads for one of the storage systems at Facebook.

1 Introduction

Almost all of the user-visible data and information served up by the Facebook app is maintained in a single directed graph called the *Social Graph* [2, 34, 35]. Friends, Checkins, Tags, Posts, Likes, and Comments are all represented as vertices and edges in the graph. As such, the graph contains billions of vertices and trillions of edges, and it consumes many hundreds of petabytes of storage space.

The information presented to Facebook users is primarily the result of dynamically generated queries on the Social Graph. For instance, a user's home profile page contains the results of hundreds of dynamically triggered queries. Given the popularity of Facebook, the Social Graph must be able to service well over a billion queries a second.

The scale of both the graph and the volume of queries makes it necessary to use a distributed system design for implementing the systems supporting the Social Graph. Designing and implementing such a system so that it operates efficiently is non-trivial.

A problem that repeatedly arises in distributed systems that serve large social networks is one of *assigning objects to components*; for example, assigning user requests to compute servers (*HTTP request routing*), or assigning data records to storage subsystems (*storage sharding*). How such assignments are made can have a significant impact on performance and resource usage. Moreover, the assignments must satisfy a wide range of requirements: e.g., they must (*i*) be amenable to quick lookup, (*ii*) respect component size constraints, and (*iii*) be able to adapt to changes in the graph, usage patterns and hardware infrastructure, while keeping the load well balanced, and (*iv*) limit the frequency of assignment changes to prevent excess overhead.

The relationship between the data of the social network and the queries on the social network is $m : n$ — a query may require several data items and a data item may be required by several queries. This makes finding a good assignment of objects to components non-trivial; finding an optimal solution for many objective functions is NP Hard [6]. Moreover, a target optimization goal, captured by an objective function, may conflict with the

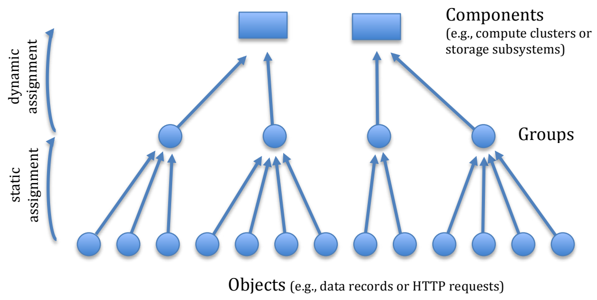

Components
(e.g., compute clusters or storage subsystems)
dynamic assignment
Groups
static assignment
Objects (e.g., data records or HTTP requests)

Figure 1: Social Hash Abstract Framework

goal of keeping the loads on the components reasonably well balanced. In the next subsection, we propose a two-level framework that allows us to trade off these two conflicting objectives.

Social Hash Framework

We have developed a general framework that accommodates the HTTP request routing and storage sharding examples mentioned above, as well as other variants of the assignment problem. In our Social Hash framework, the assignment of objects (such as users or data records) to components (such as compute clusters or storage subsystems) is done in two steps. (See Fig. 1.)

In the first step, each *object* is assigned to a *group*, where groups are conceptual entities representing clusterings of objects. Importantly, there are usually many more groups than components. This assignment is based on optimizing a given, scenario-dependent, objective function. For example, when assigning HTTP requests to compute clusters, the objective function may seek to minimize the (main memory) cache miss rate; and when assigning data records to disk subsystems, the objective function may seek to minimize the number of disk subsystems that must be contacted for multi-get queries. Because this optimization is typically computationally intensive, objects are re-assigned to groups only periodically and offline (e.g., daily or weekly). Hence, we refer to this as the *static assignment step*.

In the second step, each *group* is assigned to a *component*. This second assignment is based on inputs from system monitors and system administrators so as to rapidly and dynamically respond to changes in the system and workload. It is able to accommodate components going on or offline, and it is responsible for keeping the components' loads well balanced. Because the assignments at this level can change in real time, we refer to this as the *dynamic assignment step*.

A key attribute of our framework is the decoupling of optimization in the static assignment step, and dynamic adaptation in the dynamic assignment step. Our solutions to the assignment problem rely on being able to beneficially group together relatively small, cohesive sets of objects in the Social Graph. In the optimizations performed by the static assignment step, we use graph partitioning to extract these sets from the Social Graph or from prior access patterns. Optimization methods other than graph partitioning could be used interchangeably, but graph partitioning is expected to be particularly effective in the context of social networks, because most requests are social in nature where users that are socially close tend to consume similar data. The *Social* in Social Hash reflects this essential idea of grouping socially similar objects together.

Contributions

This paper describes the Social Hash framework for assigning objects to components given scenario-dependent optimization objectives, while satisfying the requirements of fine-grained load balancing, assignment stability, and fast lookup in the context of practical difficulties presented by changes in the workload and infrastructure.

The Social Hash framework and the two applications described in this paper have been in production use at Facebook for over a year. Over 78% of Facebook's "stateless" Web traffic routing occurs with this framework, and the storage sharding application involves tens of thousands of storage servers. The framework has also been used in other settings (e.g., to distribute vertices in a graph processing system, and to reorder data to improve compression rates). We do not describe these additional applications in this paper.

The three most important contributions we make in this paper are:

1. the two-step assignment hierarchy of our framework that decouples (*a*) optimization on the Social Graph or previous usage patterns from (*b*) adaptation to changes in the workload and hardware infrastructure;

2. our use of graph partitioning to exploit the structure of the social network to optimize HTTP routing in very large distributed systems;

3. our use of query history to construct bipartite graphs that are then partitioned to optimize storage sharding.

With respect to (1), the use of a multi-level scheme for allocating resources in distributed systems is not new, not even when used with graph partitioning [33]. In particular, some multi-tenant resource allocation schemes have used approaches that are in many respects similar to the one being proposed here [19, 26, 27, 28]. However, the specifics of our approach, especially as they relate to Facebook's operating environment and workload, are sufficiently interesting and unique to warrant a dedicated discussion and analysis. Regarding (2), edge-cut based graph partitioning techniques have been used for numerous optimization applications, but to the best of our knowledge not for making routing decisions to reduce cache miss rates. Similarly, for (3), graph partitioning has previously been applied to storage sharding [33], but partitioning bipartite graphs based on prior access patterns is, as far as we know, novel.

We show that the Social Hash framework enables significant performance improvements as measured on the production Social Graph system using live workloads. Our HTTP request routing optimization cut the cache miss rate by 25%, and our storage sharding optimization cut the average response latency in half.

2 Two motivating example applications

In this section, we provide more details of the two examples we mentioned in the Introduction. We discuss and analyze these applications in significantly greater detail in later sections.

HTTP request routing optimization. The purpose of HTTP request routing is to assign HTTP requests to compute clusters. When a cluster services a request, it fetches any required data from external storage servers, and then caches the data in a cluster-local main memory-based cache, such as TAO [2] or Memcache [24], for later reuse by other requests. For example, in a social network, a client may issue an HTTP request to generate the list of recent posts by a user's friends. The HTTP request will be routed to one of several compute clusters. The server will fetch all posts made by the user's friends from external databases and cache the fetched data. How HTTP requests are assigned to compute clusters will affect the cache hit rate (since a cached data record may be consumed by several queries). It is therefore desirable to choose a HTTP request assignment scheme which assigns requests with similar data requirements to the same compute cluster.

Storage sharding optimization. The purpose of storage sharding is to distribute a set of data records across several storage subsystems. A query which requires a certain record must communicate with the unique host that serves that record.[1] A query may consume several records, and a record may be consumed by several queries. For example, if the dataset consists of recent posts produced by all the users, a typical query might fetch the recent posts produced by a user's friends.

The assignment of data records to storage subsystems determines the number of hosts a query needs to communicate with to obtain the required data. A common optimization is to group requests destined to the same storage subsystem and issue a single request for all of them. Additionally, since requests to different storage subsystems are processed independently, they can be sent in parallel. As a result, the latency of the slowest request will determine the latency of a multi-get query, and the more hosts a query needs to communicate with, the higher the expected latency (as we show in Section 6.1). It is thus desirable to choose a data record assignment scheme that collocates the data required by similar queries within a small number of storage subsystems.

3 The assignment problem

Assigning objects to system components is a challenging part of scaling an online distributed system. In this section, we abstract the essential features of our two motivating examples to formulate the problem we solve in this paper.

3.1 Requirements

We have the following requirements:

- *Minimal average query response time*: User satisfaction can improve with low query response times.
- *Load balanced components*: The better load-balanced the components, the higher the efficiency of the system; a poorly load-balanced system will reach its capacity earlier and in some cases may lead to increased latencies.
- *Assignment stability*: Assignments of objects to components should not change too frequently in order to avoid excessive overhead. For example, reassigning a query from one cluster to another may lead to extra (cold) cache misses at the new cluster.
- *Fast lookup*: Low latency lookup of the object-component assignment is important, given the online nature of our target distributed system.

3.2 Practical challenges

Meeting the requirements listed above is challenging for a variety of reasons:

[1] To simplify our discussion, we disregard the fact that data is typically replicated across multiple storage servers.

- *Scale*: The assignment problem typically requires assigning a large number of objects to a substantially smaller number of components. The combinatorial explosion in the number of possible assignments prevents simple optimization methods from being effective.
- *Effects of similarity on load balance*: Colocating similar objects usually results in higher load imbalances than when colocating dissimilar objects. For example, similar users likely have similar hours of activity, browsing devices, and favorite product features, leading to load imbalance when assigning similar users to the same compute clusters.
- *Heterogenous and dynamic set of components*: Components are often heterogeneous and thus support different loads. Further, the desired load on each component can change over time; e.g., due to hardware failure. Finally the set of components will change over time as new hardware is introduced and old hardware removed.
- *Dynamic workload*: The relationship between data and queries can change over time. A previously rarely accessed data record could become popular, or new types of queries could start requesting data records that were previously not accessed. This can happen, for example, if friendship ties in the network are introduced or removed, or if product features change their data consumption pattern.
- *Addition and removal of objects*: Social networks change and grow constantly, so the set of objects that must be assigned changes over time. For example, users may join or leave the service.

The magnitude and relative importance of these practical challenges will differ depending on the distributed system being targeted. For Facebook, the scale is enormous; similar users do have similar patterns; and heterogeneous hardware is prevalent. On the other hand, changes to the graph occur at a (relatively) modest rate (in part because we often only consider subgraphs of the Social Graph); and rate of hardware failures is reasonably constant and predictable.

4 Social Hash Framework

In this section, we propose a framework called the Social Hash Framework which comprises a solution to the assignment problem and, moreover, addresses the practical challenges listed above.

In Section 1 we introduced the abstract framework with objects at the bottom, (abstract) groups in the middle, and components at the top. Recall that objects are queries, users, or data records, etc., and components are computer clusters, or storage subsystems, etc..

Objects are first assigned to groups in a optimization-based static assignment that is updated on a slow timescale of a day to a week. Groups are then assigned to components using an adaptation-based dynamic assignment that is updated on a much faster timescale. Dynamic assignment is used to keep the system load-balanced despite changes in the workload or changes in the underlying infrastructure. This two-level design is intended to accommodate the disparate requirements and challenges of efficiently operating a huge social network, as described in Section 3.

Below, we give more concrete details on the abstract framework, how it is implemented, and how it is used. In Sections 5 and 6 we will become even more concrete and present specific implementation issues for our two examples. We begin by presenting our rationale for using a two-level design.

4.1 Rationale

Our two-level approach for assigning objects to components is motivated by the observation that there is a conflict between the objectives of optimization and adaptation. In theory, one could assign objects to components directly, resulting in only one assignment step. However, this would not work well in practice because of difficulties adapting to changes: as mentioned, component loads often change unpredictably; components are added or removed from the system dynamically; and the similarity of objects that are naturally grouped together for optimization leads to unbalanced utilization of resources. Waiting to rerun the assignment algorithm would leave the system in a suboptimal state for too long, and changing assignments on individual objects without re-running the assignment algorithm would also be suboptimal.

An assignment framework must therefore address both the optimization and adaptation objectives, and it must offer enough flexibility to be able to shift emphasis between these competing objectives at will. With a two-level approach, the static level optimizes the assignment to groups where, from the point of view of optimization, the group is treated as a virtual component. The dynamic level adapts to changes by assigning groups to components. Multiple groups may be assigned to the same component; however, all objects in the same group are guaranteed to be assigned to the same component. (See Figure 1.) As such, what is particularly propitious about our architecture is that dynamic reassignment of groups to components does not negate the optimization step because objects in a group remain collocated to the same component, even after reassignment.

We are able to seamlessly shift emphasis between static optimization and dynamic adaptation by means of the parameter n, the ratio of number of groups to number of components; that is $n := |G| \big/ |C|$. When $n = 1$, the emphasis is entirely on the static optimization. There is a $1:1$ correspondence between groups and components. As noted above, this may not work well for some applications because it may not be sufficiently adaptive. When $n \gg 1$, we trade off optimization for increased adaptation. When n is too large, the optimization quality may be severely degraded, and the overhead of dynamic assignment may be prohibitive. Clearly, the choice of n, and thus the tradeoff between optimization and adaptation, is best selected on a per-application basis; as we show in later sections, some applications require less aggressive adaptation than others, allowing more emphasis to be placed on optimization.

4.2 Framework Overview

In this subsection, we describe the main elements of the Social Hash framework, as depicted in Fig. 2: the static assignment algorithm, the dynamic assignment algorithm, the lookup method, and the missing key assignment. In the discussion that follows it is useful to note that objects are uniquely identified by a *key*.

The static assignment algorithm generates a static mapping from objects to groups using the following input: (*i*) a context dependent graph, which in our work can be either a unipartite graph (e.g., friendship graph) or a bipartite graph based on access logs (e.g., relating queries and accessed data records); (*ii*) type of object that is to be assigned to groups (e.g. data records, users, etc); (*iii*) an objective function; (*iv*) number of groups; and (*v*) permissible imbalance between groups. The output of the static partitioning algorithm is a hash table of (*key*, *group*) pairs, indexed by *key*. We refer to this hash table as the *Social Hash Table*.[2]

The dynamic assignment uses the following input: (*i*) current component loads, (*ii*) the desired maximum load per component, and possibly (*iii*) the historical loads per group. The desired load for each component is provided by system operators and monitoring systems, and the historical loads induced by each group can be derived from system logs. As the observed and desired loads change over time, the dynamic assignment shifts groups among components to balance the load. The output of the dynamic assignment is a hash table of (*group*, *component*) pairs, called the *Assignment Table*.[2]

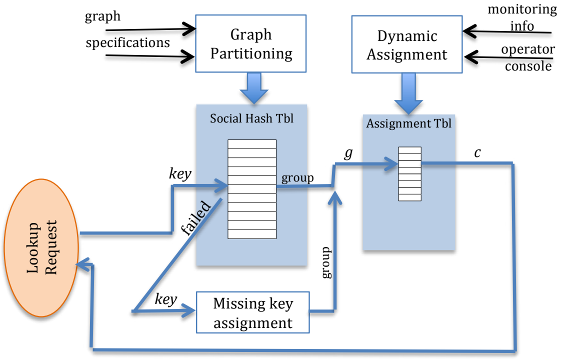


Figure 2: Social Hash Architecture

When a client wishes to look up which component an object has been assigned to, it will do so in two steps: first the object key is used to index into the Social Hash Table to obtain the target group, g; second, g is used to index into the Assignment Table to obtain the component id c. This is shown in Figure 2.

Because the Social Hash Table is constructed only periodically, it is possible that a target *key* is missing in the Social Hash table; for example, the key could refer to a new user or a user that has not had any activity in the recent past (and hence is not in the access log). When an object key is not found in the Social Hash Table, then the *Missing Key Assignment* rule does the exception handling and assigns the object to a group on the fly. The primary requirement is that these exceptional assignments are generated in a consistent way so that subsequent lookups of the same key return the same group. Eventually these new keys will be incorporated into the Social Hash Table by the static partitioning algorithm.

4.3 Static assignment algorithm

We use graph partitioning algorithms to partition objects into groups in the static assignment step. Graph partitioning algorithms have been well-studied [3], and a number of graph partitioning frameworks exist [5, 18]. However, social network graphs, like Facebook's Social Graph, can be huge compared to what much of the existing literature contemplates. As a result, an approach is needed that is amenable to distributed computation on distributed memory systems. We built our custom graph partitioning solution on top of the Apache Giraph graph processing system [1], in part because of its ability to partition graphs in parallel; other graph processing systems could have also potentially been used [10, 11, 20].

The basic strategy in obtaining good static assignments is the following graph partitioning heuristic. We

[2] In practice, any key-value store that supports fast lookups can be used. We describe it as a hash table for ease of comprehension.

assume the algorithm begins with an initial (weight-) balanced assignment of objects to groups represented as pairs (v, g_v), where v denotes an object and g_v denotes the group to which v is initially assigned. Next, for each v, we record the group g^*_v that gives the optimal assignment for v to minimize the objective function, assuming all other assignments remain the same. This step is repeated for each object to obtain a list of pairs (v, g^*_v). Each object can be processed in parallel. Finally, via a swapping algorithm, as many reassignments of v to g^*_v are carried out under the constraint that group sizes remain unchanged within each iteration; the swapping can again be done in parallel as long as it is properly coordinated (in our implementation with a centralized coordinator). This overall process is then repeated with the new assignments taken as the initial condition for the next iteration. The above process is iterated on until it converges or reaches a limit on the number of iterations.

The initial balanced assignment required by the static assignment algorithm is either obtained via a random assignment (e.g., when the algorithm is run for the very first time) or is obtained from the output of the previous iteration of the static assignment algorithm modulo the newly added objects that are assigned randomly.

The above procedure manages to produce high quality results for the graphs underlying Facebook operations in a fast and scalable manner. Within a day, a small cluster of a few hundred machines is able to partition the friendship graph of over 1.5B+ Facebook users into 21,000 balanced groups such that each user shares her group with at least 50% of her friends. And the same cluster is able to update the assignment starting from the previous assignment within a few hours, easily allowing a weekly (or even daily) update schedule. Finally, it is worth pointing out that the procedure is able to partition the graph into tens of thousands of groups, and it is amenable to maintaining stability, since each iteration begins with the previous assignment and it is easy to limit the movement of objects across groups.

We have successfully used the above heuristic on both unipartite and bipartite graphs, as we describe in more detail in Sections 5 and 6.

4.4 Dynamic assignment

The primary objective of dynamic assignment is to keep component loads well balanced despite changes in access patterns and infrastructure. Load balancing has been well researched in many domains. However, the specific load balancing strategy used for our Social Hash framework may vary from application to application so as to provide the best results. Factors that may affect the the choice of load balancing strategy include:

- *Accuracy in predicting future loads*: Low prediction accuracy favors a strategy with a high group-to-component ratio (e.g., $\gg 1,000$) and groups being assigned to components randomly. This is the strategy that is used for HTTP routing. On the other hand, the amount of storage used by data records is easier to predict (in our case), and hence warrants a low group-to-component ratio and non-random component assignment.
- *Dimensionality of loads*: A system requiring balancing across multiple different load dimensions (CPU, storage, queries per second, etc.) favors using a high group-to-component ratio and random assignment.
- *Group transfer overhead*: The higher the overhead of moving a group from one component to another, the more one would want to limit the rate of moves between components by increasing the load imbalance threshold that triggers a move.
- *Assignment memory*: It can be more efficient to assign a group back to an underloaded component it was previously assigned to in order to potentially benefit from the residual state that may still be present. This favors remembering recent assignments, or using techniques similar to consistent hashing.

Finally, we note that load balancing strategies used in other domains will need to be adapted to the Social Hash framework; e.g., load is transferred from one component to another in increments of a group; and the load each group incurs is not homogeneous, in part because of the similarity of objects within groups.

5 Social Hash for Facebook's Web Traffic Routing

In this section, we describe how we applied the Social Hash framework to Facebook's global web traffic routing to improve the efficiency of large cache services. This is Facebook's largest application using the framework and has been in production for over a year.

Facebook operates several worldwide data centers, each divided into front-end clusters containing web and cache tiers, and back-end clusters containing database and service tiers. To fulfill an HTTP request, a front-end web server may need to access databases or services in (possibly remote) back-end clusters. The returned data is cached within front-end cache services, such as TAO [2] or Memcache [24]. Clearly, the lower the cache miss rate, the higher the efficiency of hardware usage, and the lower the response times.

In addition, to reduce latencies for users, Facebook has "Point-of-Presence" (PoP) units around the world: small-scale computational centers which reside close to users. PoPs are used for multiple purposes, including peering with other network operators and media caching [12]. When an HTTP request to one of Facebook's services is issued, the request will first go to a nearby PoP. A load balancer in the PoP then routes the request to one of the front-end clusters over fast communication channels.

5.1 Prior strategy

Prior to using Social Hash, routing decisions were based on user identifiers, using a consistent hashing scheme [15]. To make a routing decision, the user identifier was extracted from the request, where it was encoded within the request persistent attributes (i.e., cookie), and then used to index into a consistent hash ring to obtain the cluster id. The segments of the consistent hash ring corresponded in number and weight to the front-end clusters. The ring's weights were dynamic and could be changed at any time, allowing dynamic changes to the cluster's traffic load. The large number of users in comparison to the small number of clusters, along with the random nature of the hash ring, ensured that each cluster received a homogeneous traffic pattern. With fixed cluster weights, a user would repeatedly be routed to the same cluster, guaranteeing high hit rates for user-specific data. The consistent nature of the ring also ensured that changes to cluster weights resulted in relatively minor changes to the user-to-cluster mapping, reducing the number of cache misses after such changes.

5.2 Social Hash implementation

For the Social Hash static assignment, we used a unipartite graph with vertices representing Facebook's users and edges representing the friendship ties between them. We partition the graph using the edge-cut optimization criterion, knowing that friends and socially similar users tend to consume the same data, and that they are therefore likely to reuse each other's cached data records.

We use a relatively large number of groups for two reasons. First, the global routing scheme needs to be able to shift traffic across clusters in small quantities. Second, changes in HTTP request routing will affect many subsystems at Facebook, not just the cache tiers; and it is very difficult to predict how much load each group will incur on each subsystem. Hence, we have found the best strategy to balance the load overall is to use many groups and assign the groups to clusters randomly.

For the dynamic assignment step, we kept the existing consistent hash scheme, which is oblivious to the type of identifier it receives as input (either user- or group-id).

To be able to make an HTTP request routing decision at run time, it is necessary to access both the Social Hash Table and the Assignment Table. The latter is computed on-the-fly using the consistent hash mechanism, which requires a fairly small map between clusters and their weights; it is therefore easy to hold the map in the POP memories. The former, however, is large, consuming several gigabytes of storage space when uncompressed. We considered storing the Social Hash Table close to the PoP (in its own memory or in a nearby storage service), but decided not to do so due to added PoP complexity, fault tolerance considerations, and limited PoP resources that could be put to better use by other services. We also considered sending a lookup request to a data center, but rejected this idea due to latency concerns.

Instead, we encode the user assigned group within the request persistent attributes (i.e., as a cookie) and decode it to make a routing decision when a request arrives. Requests that do not have the group-id encoded in the header are routed to a random front-end cluster, where the session creation mechanism accesses a local copy of the Social Hash Table to fetch the group assigned to the user. Because the Social Hash Table is updated once a week, group-ids in the headers may become stale. For this reason, a user request will periodically (at least once an hour) trigger an update process where the group-id is updated with its latest value from the Social Hash Table. This allows long lasting connections to receive fresh routing information.

Our design eliminates the complexities and overhead of a Social Hash Table lookup at the PoPs, requiring just a single header read instead. The design is also more resilient to failure, because even if the data store providing the Social Hash Table is down, group-id's will mostly be available in the request headers.

For technical reasons, some requests cannot be tagged properly with either the user or the group identifier (because the requests may have been issued by crawlers, bots or legacy clients). These requests are routed randomly, yet in a consistent manner, to one of the front-end clusters while respecting load constraints. In the past three months, 78% of the requests had a valid group-id that could be used for routing (and those that did not were not tagged with a user-id, a group-id, or any other identifier.).

Some may argue that the decreased miss rates achieved with Social Hash leads to a fault tolerance issue, because the data records are less likely to be present in

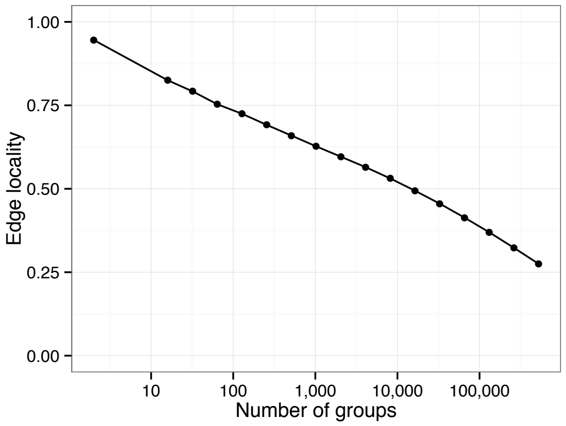


Figure 3: Edge locality (fraction of edges within groups) vs. the number of groups for Facebook's friendship graph.

multiple caches simultaneously. This could be a concern as the recovery of a cache failure would overwhelm the backing storage systems with excessive traffic and thus lead to severely degraded overall performance. However, our experience indicates that a failure of the main-memory caches within a cluster only causes a temporary load increase on the backend storage servers that stays within the normal operational load thresholds.

5.3 Operational observations

To get a sense of how access patterns of friends relate, we sampled well over 100 million accesses to TAO data records from access logs. We found that when two users access the same data record, there is a 15% chance they are friends. This is millions of times larger than the probability of two random users being friends. We conclude that co-locating the processing of friends' HTTP requests as much as possible is an effective strategy.

Figure 3 depicts edge locality vs. the number of groups used to partition the 1.5B+ Facebook users. Edge locality measures the fraction of "friend" edges connecting two users that are both assigned to the same group (thus, the goal of static assignment would be to maximize edge locality). It is not a surprise that edge locality decreases with the number of groups. Perhaps a bit more unexpected is the observation that edge locality remains reasonably large even when the number of groups increases significantly (e.g., >20% with 1 million groups); intuitively, this is because the friendship graph contains many small relatively dense communities. We chose the smallest number of groups that would satisfy our main requirement for dynamic assignment, namely to be able to balance the load by shifting only small amounts of traffic between front-end clusters. Repeating the process of assigning different numbers of groups into components offline and examining the resulting imbalance on known loads led us to use 21,000 groups on a few 10's of clusters; our group-to-component ratio is thus quite high.

The combination of new users being added to the system and changes to the friendship graph causes edge locality to degrade over time. We measured the decrease of edge locality from an initial, random assignment of users to one of 21,000 groups over the course of four weeks. We observed a nearly linear 0.1% decrease in edge locality per week. While small, we decided to update the routing assignment once a week so as to minimize a noticeable decrease in quality. At the same time, we did not observed a decrease in cache hit rate between updates, implying that 0.1% is a negligible amount. The decrease in edge locality implies that a longer update schedule would also be satisfactory, and that Social Hash can tolerate a long maintenance breakdown without altering Facebook's web traffic routing quality.

For the past three months, the Social Hash Table used for Facebook's routing has maintained an edge locality of over 50%, meaning half the friendships are within each of the 21,000 groups. This edge locality is slightly higher than the exploratory values shown in Figure 3, because we iterated longer in the graph partitioning algorithm on the production system than we did in the experiments from which we obtained the figure. The static assignment is well-balanced, with the largest group containing at most 0.8% more users than the average group. Each weekly update by the static assignment step resulted in around 1.5% of users switching groups from the previous assignment. All of these updates were suitably small to avoid noticeable increases in the cache miss rate when the updates were introduced into production.

5.4 Live traffic experiment

To measure the effectiveness of Social Hash-based HTTP routing optimization, we performed a live traffic experiment on two identical clusters with the same hardware, number of hosts and capacity constraints. These clusters are typical of what Facebook uses in production. Each cluster had many hundred TAO servers, which served the local web tier with cached social data.

For our experiment, we selected a set of groups randomly from the Social Hash Table. We then routed all HTTP requests from users assigned to these groups to one "test" cluster, while HTTP requests from a same number of other users were routed to the second, "control" cluster. Hence, the control cluster saw traffic with

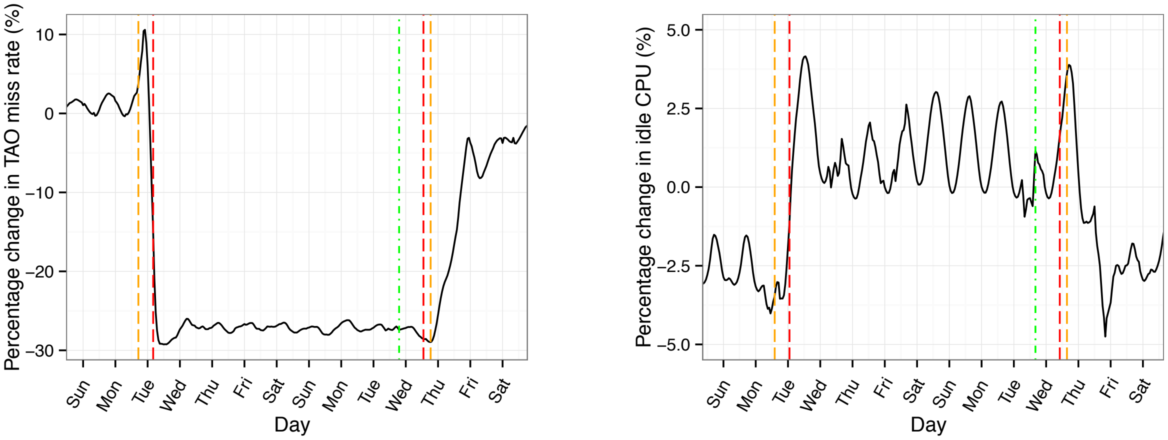


Figure 4: Percentage change in TAO miss rate (left, where lower is better) and CPU idle rate (right, where higher is better) on the Social Hash cluster relative to the cluster with random assignment. Area between red dashed lines: period of the test. Orange dashed lines: traffic shifts. Green dot-dash line: Social Hash Table is updated. The values on the days traffic was shifted (Tuesday and Wednesday, respectively) are not representative

attributes very similar to the traffic it received with the prior strategy: the traffic with the prior strategy was sampled from all users, while the traffic for the control cluster was sampled from all users except those associated with the test cluster. We ran the experiment for 10 days. During this time, operational changes that would affect hit rates on the two clusters were prevented.

The left hand side of Figure 4 shows the change in cache miss rate between the test and control clusters. It is evident that the miss rate drops by over 25% when assigning groups to a cluster as opposed to just users.

The right hand side of Figure 4 shows the change in average CPU idle rate between the test and the control cluster. The test cluster had up to 3% more idle time compared to the control cluster.

During the experiment, we updated the Social Hash Table by applying an updated static assignment. The time at which this occurred is shown with a vertical green dot-dash line. We note that the cache miss rate and the CPU idle time are not affected by the update, demonstrating that the transition process is smooth.

Figure 5 compares the daily working set size for TAO objects at both clusters. The daily working set of a cluster is the total size of all objects that were accessed by the TAO instance on that front-end cluster at least once during that day. The figure shows that the working set size dropped by as much as 8.3%.

We conclude from this experiment that Social Hash is effective at improving the efficiency of the cache for HTTP requests: fewer requests are sent to backend systems, and the hardware is utilized in a more efficient way.

6 Storage sharding

In this section, we describe in detail how we applied the Social Hash framework to sharded storage systems at Facebook. The assignment problem is to decide how to assign data records (the objects) to storage subsystems (the components).

6.1 Fanout vs. Latency

The objective function we optimize is *fanout*, the number of storage subsystems that must be contacted for multiget queries. We argue and experimentally demonstrate that fanout is a suitable objective function, since lower fanout is closely correlated with lower latencies [7].

Multiget queries are typically forced to issue requests to multiple storage subsystems, and they do so in parallel. As such, the latency of a multi-get query is determined by the slowest request. By reducing fanout, the probability of encountering a request that is unexpectedly slower than the others is reduced, thus reducing the latency of the query. This is the fundamental argument for using fanout as the objective function for the assignment problem in the context of storage sharding. Another argument is that lower fanout reduces the connection overhead per data record.

To further elaborate the relevance of choosing fanout as the objective function, consider this abstract scenario. Suppose 1% of individual requests to storage servers incur a significant delay due to unanticipated system-specific issues (CPU thread scheduling delays, system

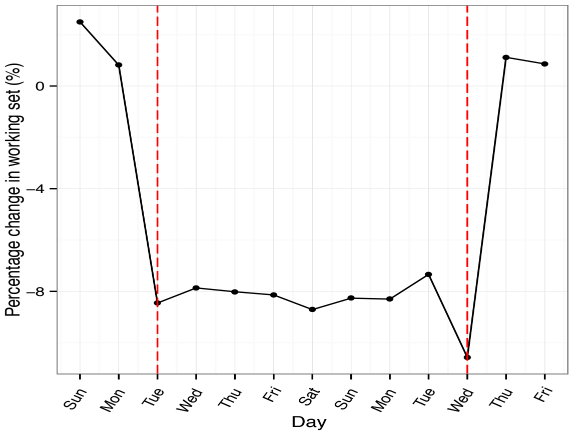


Figure 5: Percentage change in daily TAO working set size on the Social Hash cluster relative to the cluster with random assignment. The red dashed lines indicate the first and last days of the test where the test was running only during part of the day (so the values for these two days may not be representative).

interrupts, etc.). If a query must contact 10 storage servers, then one can calculate that the multi-get request has a 9.6% chance an individual sub-request will experience a significant delay. If the fanout can be reduced, one can reduce the probability of incurring the delay.

We ran a simple experiment to confirm our understanding of the relationship between fanout and latency. We issued trivial remote requests and measured the latency of a single request (fanout=1) and the latency of two requests sent in parallel (fanout=2). Figure 6 shows the cumulative latency distribution for both cases. A fanout of 1 results in lower latencies than a fanout of 2. If we calculate the expected distribution computed from two independent samples from the single request distribution, then the observed overall latency for two parallel requests matches the expected distribution quite nicely.

One possible caveat to our analysis of the relationship between fanout and latency is that reducing fanout generally increases the size of the largest request, which could increase latency. Fortunately, storage subsystems today have processors with many cores that can be exploited by the software to increase the parallelism in servicing a single, large request.

6.2 Implementation

For the static assignment we apply bipartite graph partitioning to minimize fanout. We create the bipartite graph from logs of queries from the dynamic operations of the

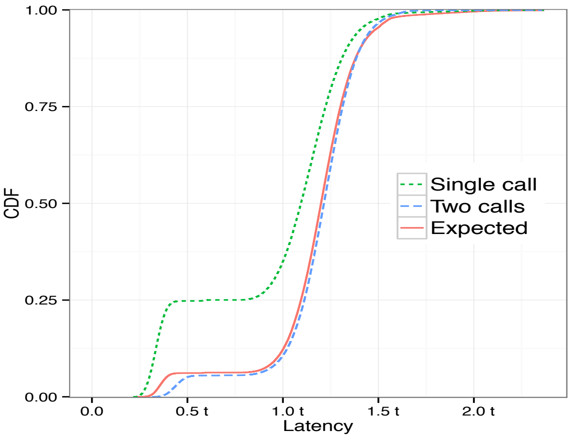


Figure 6: Cumulative distribution of latency for a single request, two requests in parallel, and the expected distribution from two independent samples from the single request distribution, where t is the average latency of a single call

.

social network.[3] The queries and data records accessed by the queries are represented by two types of vertices. A query vertex is edge-connected to a data vertex iff the query accesses the data record. The graph partitioning algorithm is then used to partition the data vertices into groups so as to minimize the average number of groups each query is connected to.

Clearly, most data needs to be replicated for fault tolerance (and other) reasons. Many systems at Facebook do this by organizing machines storing data into non-overlapping sets, each containing each data record exactly once. We refer to such a set as a replica. Since assignment is independent between replicas, we will restrict our analysis to scenarios with just one replica.

6.3 Simplified sharding experiment

We consider the following simple experiment. We use 40 stripped down storage servers, where data is stored in a memory-based, key-value store. We assume that there is one data record per user. We run this setup in two configurations. In the first, “random” configuration, data records are distributed across the 40 storage servers using a hash function, which is a common practice. In the second, “social” configuration, we use our Social Hash framework to minimize fanout.

We then sampled a live traffic pattern, and issued the same set of queries to both configurations, and we measured fanout and latency. With the random configuration,

[3] In some cases, prior knowledge of which records each query must retrieve is sufficient to create the graph.

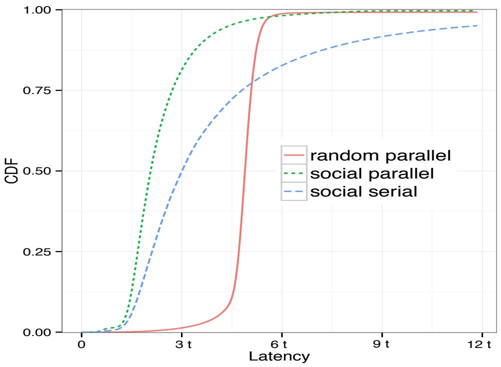


Figure 7: Cumulative latency distribution for fetching data of friends, where t is the average latency of a single call.

the queries needed to issue requests to 38.8 storage subsystems on average. With the social configuration, the queries needed to issue requests to only 9.9 storage subsystems on average. This decrease in fanout resulted in a 2.1X lower average latency for the queries.

The cumulative distribution of latencies for the random and social configurations are shown in Figure 7, where we also include the social configuration's latency distribution after disabling parallelism within each machine. Without parallelism, the average latency is still lower then with the random configuration, but only by 23%. Furthermore, the slowest 25% queries on the social configuration without parallelism exhibited substantially higher latencies than the 25% slowest queries on the random configuration. This figure confirms the importance of using parallelism within each system.

6.4 Operational observations

After we deployed storage sharding optimized with Social Hash to one of the graph databases at Facebook, containing thousands of storage servers, we found that measured latencies of queries decreased by over 50% on average, and CPU utilization also decreased by over 50%.

We attribute much of this improvement in performance to our method of assigning data records to groups, using graph partitioning on bi-partite graphs generated from prior queries. The solid line in Figure 8 shows the average fanout as a function of the number of groups when using our method. The dotted line shows the average fanout when using standard edge-cut optimization criteria on the (unipartite) friendship graph.

After analyzing expected load balance, we decided on a group-to-component ratio of 8; the dynamic assignment algorithm then selects which 8 groups to assign to the same storage subsystem, based on the historical load patterns. This allowed us to keep fanout small, while still being able to maintain good load balance.

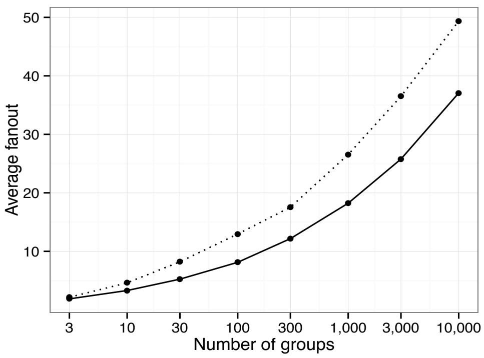


Figure 8: The average fanout versus number of groups on Facebook's friendship graph when using edge locality optimization (dotted curve) and our fanout optimization (solid curve), respectively.

In practice, fanout degrades over time. For the 40 group solution we used in the simplified application, we observed a fanout increase of about 2% on average over the course of a week. A single static assignment update sufficed to bring the fanout back to what it was previously, requiring only 1.85% of the data records to have to be moved. With such low impact, we decided static assignment updates were only necessary every few months, relying on dynamic assignment to move groups when necessary in between. Even then, we found that dynamic assignment updates were not necessary more than once a week on average. We used the same static assignment for all replicas, but made dynamic assignment decisions independently for each replica.

7 Related work

As discussed in Section 4.3, graph partitioning has an extensive literature, and our optimization objectives, edge locality and fanout, correspond to edge cut and hypergraph edge cut. A recent review of graph partitioning methods can be found online [3]. Many graph partitioning systems have been built and are available. For example, Metis [16, 18] is one that is frequently used.

A Giraph-based approach to graph partitioning called "Spinner" was recently announced [21]. Our work is distinct in that their application was optimizing batch processing systems, such as Giraph itself, via increased edge locality, and our graph partitioning system is embedded in the Social Hash framework.

Average fanout in a bipartite graph, when presented as a hypergraph, with vertices being one side of the bipartite graph, and hyper-edges representing the vertices from the other side, directly translates into the hypergraph partitioning problem. Hypergraph partitioning also has an extensive literature [4, 17], and one of the publicly available parallel solutions is PHG [9], which can be found in the Zoltan toolkit [8].

Partitioning online social networks has previously been used to improve performance of distributed systems. Ugander and Backstrom discuss partitioning large graphs to optimize Facebook infrastructure [33]. Stein considered a theoretical application of partitioning to Facebook infrastructure [29]. Tran and Zhang considered a multi-objective optimization problem based on edge cut motivated by read and write behaviors in online social networks [31, 32].

Other research has considered data replication in combination with partitioning for sharding data for online social networks. Pujol et al. studied low fanout configurations via replication of data between hosts [25] and Wang et al. suggested minimizing fan-out by random replication and query optimization [36]. Nguyen et al. considered how to place additional replicas of users given a fixed initial assignment of users to servers [22, 30].

Dean and Barroso [7] investigated the effect of latency variability on fanout queries in distributed systems, and suggested several means to reduce its influence. Jeon et al. [14] argued for the necessity of parallelizing execution of large requests, in order to tame latencies.

Our contribution differs from these lines of research by presenting a realized framework integrated into production systems at Facebook. A production application to online social networks is provided by Huang et al. who describe improving infrastructure performance for Renren through a combination of graph partitioning and data replication methods [13]. Sharding has been considered for distributed social network databases by Nicoara, et al. who propose Hermes [23].

8 Concluding Remarks

We introduced the Social Hash framework for producing, serving, and maintaining assignments of objects to components in distributed systems. The framework was designed for optimizing operations on large social networks, such as Facebook's Social Graph. A key aspect of the framework is how optimization is decoupled from dynamic adaptation, through a two-level scheme that uses graph partitioning for optimization at the first level and dynamic assignment at the second level. The first level leverages the structure of the social network and its usage patterns, while the second level adapts to changes in the data, its access patterns and the infrastructure.

We demonstrated the effectiveness of the Social Hash framework with the HTTP request routing and storage sharding applications. For the former, Social Hash was able to decrease the cache miss rate by 25%, and for the latter, it was able to cut the average response time in half, as measured on the live Facebook system with live traffic production workloads. The approaches we took with both applications was, to the best of our knowledge, novel; i.e., graph partitioning the Social Graph to optimize HTTP request routing, and using query history to construct bipartite graphs that are then partitioned to optimize storage sharding.

Our approach has some limitations. It was designed in the context of optimizing online social networks and hence will not be suitable for every distributed system. To be successful, both the static and dynamic assignment steps rely on certain characteristics, which tend to be fulfilled by social networks. For the static step, the underlying graph must be conducive to partitioning, and the graph must be reasonably sparse so that the partitioning is computationally tractable; social graphs almost always meet those characteristics. The social graph cannot be changing too rapidly; otherwise the optimized static assignment will be obsolete too quickly and the attendant exception handling becomes too computationally complex. For the dynamic step, we assume that the workload and the infrastructure does not change too rapidly.

While we have been able to obtain impressive efficiency gains using the Social Hash framework, we believe there is much room for further improvement. We are currently: (*i*) working on improving the performance of our graph partitioning algorithms, (*ii*) considering using historical query patterns and bi-partite graph partitioning to further improve cache miss rates, (*iii*) incorporating geo-locality considerations for our HTTP routing optimizations, and (*iv*) incorporating alternative replication schemes for further reducing fanout in storage sharded systems.

Acknowledgements

We would like to thank Tony Savor, Kenny Lau, Venkat Venkataramani and Avery Ching for their support, Alex Laslavic, Praveen Kumar, Jan Jezabek, Alexander Ramirez, Omry Yadan, Michael Paleczny, Jianming Wu, Chunhui Zhu, Deyang Zhao and Pavan Athivarapu for helping integrate with Facebook systems, Sanjeev Kumar, Kaushik Veeraraghavan, Dionysis Logothetis, Romain Thibaux and Rajesh Nishtala for their feedback on early drafts, Badr El-Said and Laxman Dhulipala for their contributions to the framework, and Dimitris Achlioptas for discussions on graph partitioning. We would also like to thank the reviewers for their constructive and helpful comments.

References

[1] Apache Giraph. http://giraph.apache.org/.

[2] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the Social Graph. In *Proc. 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 49–60, 2013.

[3] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. *CoRR*, abs/1311.3144, 2013.

[4] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[5] C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *CoRR*, abs/0907.1375, 2009.

[6] R. Cohen, L. Katzi, and D. Raz. An efficient approximation for the generalized assignment problem. *Information Processing Letters*, 100(4):162–166, 2006.

[7] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2):74–80, Feb. 2013.

[8] K. Devine, E. Boman, L. Riesen, U. Catalyurek, and C. Chevalier. Getting started with Zoltan: A short tutorial. In *Proc. 2009 Dagstuhl Seminar on Combinatorial Scientific Computing*, 2009. Also available as Sandia National Labs Tech Report SAND2009-0578C.

[9] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proc. Intl. Parallel and Distributed Processing Symposium (IPDPS)*, pages 10–20, 2006.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. 10th Symp. on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[11] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proc. 11th Symp. on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, Oct. 2014.

[12] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *Proc. 24th Symp. on Operating Systems Principles (SOSP'13*.

[13] Y. Huang, Q. Deng, and Y. Zhu. Differentiating your friends for scaling online social networks. In *Proc. IEEE Intl. Conf. on Cluster Computing (CLUSTER'12)*, pages 411–419, Sept 2012.

[14] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: Taming tail latencies in Web search. In *Proc. 37th Intl. ACM SIGIR Conference on Research & Development in Information Retrieval (SIGIR'14)*, pages 253–262, 2014.

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. 29th Annual ACM Symp. on Theory of Computing (STOC'97)*, pages 654–663, 1997.

[16] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[17] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3):285–300, 2000.

[18] D. Lasalle and G. Karypis. Multi-threaded graph partitioning. In *Proc. IEEE 27th Intl. Symp. on Parallel and Distributed Processing (IPDPS'13)*, pages 225–236, 2013.

[19] H. Lin, K. Sun, S. Zhao, and Y. Han. Feedback-control-based performance regulation for multi-tenant applications. In *Proc. 15th Intl. Conf. on Parallel and Distributed Systems (ICPADS'09)*, pages 134–141, Dec 2009.

[20] 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 *Proc. ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'10)*, pages 135–146, 2010.

[21] C. Martella, D. Logothetis, and G. Siganos. Spinner: Scalable graph partitioning for the cloud. *CoRR*, abs/1404.3861, 2014.

[22] K. Nguyen, C. Pham, D. Tran, F. Zhang, et al. Preserving social locality in data replication for online social networks. In *Proc. 31st Intl. Conf. on Distributed Computing Systems Workshops (ICDCSW)*, pages 129–133, 2011.

[23] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *Proc. 18th Intl. Conf. on Extending Database Technology (EDBT'15)*, pages 25–36, Mar. 2015.

[24] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX Conf. on Networked Systems Design and Implementation (NSDI'13)*, pages 385–398, 2013.

[25] 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. *SIGCOMM ComputĊommun. Rev.*, 40(4):375–386, Aug. 2010.

[26] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proc. 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI 12)*, pages 349–362, 2012.

[27] D. D. C. Shue. *Multi-tenant Resource Allocation For Shared Cloud Storage*. PhD thesis, Princeton University, 2014.

[28] Y. Song, Y. Sun, and W. Shi. A two-tiered on-demand resource allocation mechanism for VM-based data centers. *IEEE Trans. on Services Computing*, 6(1):116–129, 2013.

[29] D. Stein. Partitioning social networks for data locality on a memory budget. Master's thesis, University of Illinois, Urbana-Champaign, 2012.

[30] D. A. Tran, K. Nguyen, and C. Pham. S-CLONE: Socially-aware data replication for social networks. *Computer Networks*, 56(7):2001–2013, 2012.

[31] D. A. Tran and T. Zhang. Socially aware data partitioning for distributed storage of social data. In *Proc. IFIP Networking Conference*, pages 1–9, May 2013.

[32] D. A. Tran and T. Zhang. S-PUT: An EA-based framework for socially aware data partitioning. *Computer Networks*, 75:504–518, Dec. 2014.

[33] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proc. 6th ACM Intl. Conf. on Web Search and Data Mining (WSDM-13)*, pages 507–516, 2013.

[34] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the Facebook Social Graph. *CoRR*, abs/1111.4503, 2011.

[35] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, et al. TAO: How Facebook serves the Social Graph. In *Proc. 2012 ACM SIGMOD Intl. Conf. on Management of Data*, pages 791–792, 2012.

[36] R. Wang, C. Conrad, and S. Shah. Using set cover to optimize a large-scale low latency distributed graph. In *Proc 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

The Design and Implementation of the Warp Transactional Filesystem

Robert Escriva, Emin Gün Sirer
Computer Science Department, Cornell University

Abstract

This paper introduces the Warp Transactional Filesystem (WTF), a novel, transactional, POSIX-compatible filesystem based on a new *file slicing* API that enables efficient zero-copy file transformations. WTF provides transactional access spanning multiple files in a distributed filesystem. Further, the file slicing API enables applications to construct files from the contents of other files without having to rewrite or relocate data. Combined, these enable a new class of high-performance applications. Experiments show that WTF can qualitatively outperform the industry-standard HDFS distributed filesystem, up to a factor of four in a sorting benchmark, by reducing I/O costs. Microbenchmarks indicate that the new features of WTF impose only a modest overhead on top of the POSIX-compatible API.

1 Introduction

Distributed filesystems are a cornerstone of modern data processing applications. Key-value stores such as Google's BigTable [11] and Spanner [14], and Apache's HBase [7] use distributed filesystems for their underlying storage. MapReduce [15] uses a distributed filesystem to store the inputs, outputs, and intermediary processing steps for offline processing applications. Infrastructure such as Amazon's EBS [2] and Microsoft's Blizzard [28] use distributed filesystems to provide storage for virtual machines and cloud-oblivious applications.

Yet, current distributed filesystems exhibit a tension between retaining the familiar semantics of local filesystems and achieving high performance in the distributed setting. Often, designs will compromise consistency for performance, require special hardware, or artificially restrict the filesystem interface. For example, in GFS, operations can be inconsistent or, "consistent, but undefined," even in the absence of failures [19]. GFS-backed applications must account for these anomalies, leading to additional work for application programmers. HDFS [4] side-steps this complexity by prohibiting concurrent or non-sequential modifications to files. This obviates the need to worry about nuances in filesystem behavior, but fails to support use cases requiring concurrency or random-access writes. Flat Datacenter Storage [29] is eventually consistent and requires a network with full-bisection bandwidth, which can be cost prohibitive and is not possible in all environments.

This paper introduces the Warp Transactional Filesystem (WTF), a new distributed filesystem that exposes transactional support with a new API that provides *file slicing* operations. A WTF transaction may span multiple files and is fully general; applications can include calls such as read, write, and seek within their transaction. This file slicing API enables applications to efficiently read, write, and rearrange files without rewriting the underlying data. For example, applications may concatenate multiple files without reading them; garbage collect and compress a database without writing the data; and even sort the contents of record-oriented files without rewriting the files' contents.

The key design decision that enables WTF's advanced feature set is an architecture that represents filesystem data and metadata to ensure that filesystem-level transactions may be performed using, solely, transactional operations on metadata. Custom storage servers hold filesystem data and handle the bulk of I/O requests. These servers retain no information about the structure of the filesystem; instead, they treat all data as opaque, immutable, variable-length arrays of bytes, called *slices*. WTF stores references to these slices in HyperDex [17] alongside metadata that describes how to combine the slices to reconstruct files' contents. This structure enables bookkeeping to be done entirely at the metadata level, within the scope of HyperDex transactions.

Supporting this architecture is a custom concurrency control layer that decouples WTF transactions from the underlying HyperDex transactions. This layer ensures that transactions only abort when concurrently-executing transactions change the filesystem and generate an application-visible conflict. This seemingly minor functionality enables WTF to support concurrent operations with minimal abort-induced overheads.

Overall, this paper makes three contributions. First, it describes a new API for filesystems called file slicing that enables efficient file transformations. Second, it describes an implementation of a transactional filesystem with minimal overhead. Finally, it evaluates WTF and the file slicing interfaces, and compares them to the non-transactional HDFS filesystem.

2 Design

WTF's distributed architecture consists of four components: the metadata storage, the storage servers, the replicated coordinator, and the client library. Figure 1 summarizes this architecture. The metadata storage builds on top of HyperDex and its expansive API. The storage servers hold filesystem data, and are provisioned for high I/O workloads. A replicated coordinator service serves as a rendezvous point for all components of the system, and maintains the list of storage servers. The client library contains the majority of the functionality of the system, and is where WTF combines the metadata and data into a coherent filesystem.

In this section, we first explore the file slicing abstraction to understand how the different components contribute to the overall design. We will then look at the design of the storage servers to understand how the system stores the majority of the filesystem information. Finally, we discuss performance optimizations and additional functionality that make WTF practical, but are not essential to the core design, such as replication, fault tolerance, and garbage collection.

2.1 The File Slicing Abstraction

WTF represents a file as a sequence of byte arrays that, when overlaid, comprise the file's contents. The central abstraction is a *slice*, an immutable, byte-addressable, arbitrarily sized sequence of bytes. A file in WTF, then is a sequence of slices and their associated offsets. This representation has some inherent advantages over block-based designs. Specifically, the abstraction provides a separation between metadata and data that enables filesystem-level transactions to be implemented using, solely, transactions over the metadata. Data is stored in the slices, while the metadata is a sequence of slices. WTF can transactionally change these sequences to change the files they represent, without rewriting data.

Concretely, file metadata consists of a list of *slice pointers* that indicate the exact location on the storage servers of each slice. A slice pointer is a tuple consisting of the unique identifier for the storage server holding the slice, the local filename containing the slice on that storage server, the offset of the slice within the file, and the length of the slice. Associated with each slice pointer is an integer offset that indicates where the slice should be overlaid when reconstructing the file. Crucially, this representation is self-contained: everything necessary to retrieve the slice from the storage server is present in the slice pointer, with no need for extra bookkeeping elsewhere in the system. As we will discuss later, the metadata also contains standard info found in an inode, such as modification time, and file length.

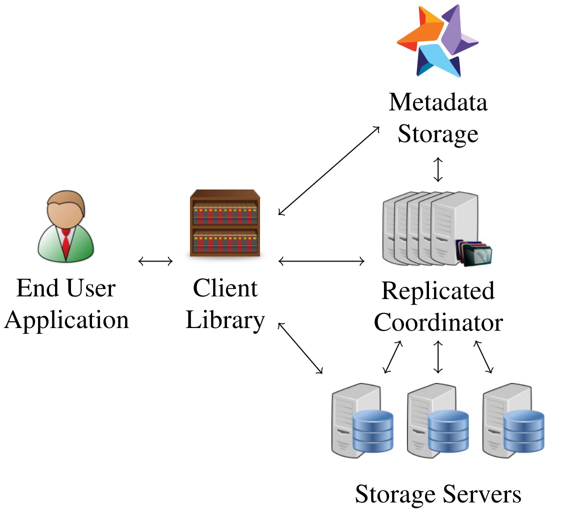
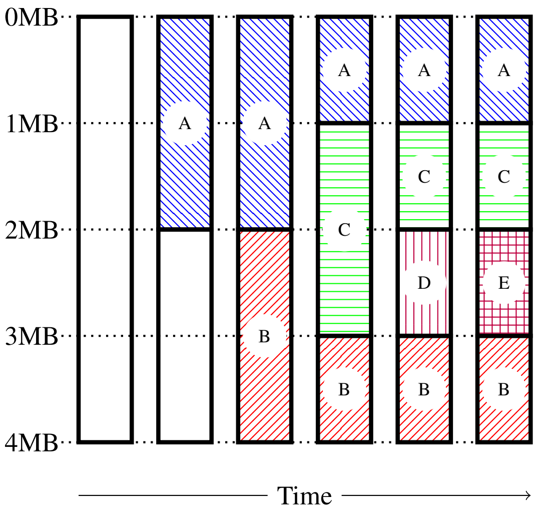


Figure 1: WTF employs a distributed architecture consisting of metadata storage, data storage, a replicated coordinator, and the client library. The client library unifies the metadata storage and storage servers to provide a filesystem interface.

This slice pointer representation enables WTF to easily generate new slice pointers that refer to subsequences of existing slices. Because the representation directly reflects the global location of a slice on disk, WTF may use simple arithmetic to create new slice pointers.

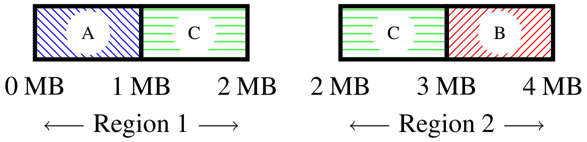
This representation also enables applications to modify a file with only localized modifications to the metadata. Figure 2 shows an example file consisting of five different slices. Each slice is overlaid on top of previous slices. Where slices overlap, the latest additions to the metadata take precedence. For example, slice C takes precedence over slices A and B; similarly, slice E completely obscures slice D and part of C. The file, then, consists of the corresponding slices of A, C, E, and B. The figure also shows the *compacted* metadata for the same file. This compacted form contains the minimal slice pointers necessary to reconstruct the file without reading data that is hidden by another slice. Crucially, all file modifications can be performed by appending to the list of slice pointers.

The procedures for reading and writing follow directly from the abstraction. A writer creates one or more slices on the storage servers, and overlays them at the appropriate positions within the file by appending their slice



Final Metadata:
A@[0,2], B@[2,4], C@[1,3], D@[2,3], E@[2,3]
Compacted Final Metadata:
A@[0,1], C@[1,2], E@[2,3], B@[3,4]

Figure 2: Writers append to the metadata list to change the file. Every prefix of the shown metadata list represents a valid state of the file at some point in time. The compacted metadata occupies less space by rearranging the metadata list to remove overwritten data.



Region 1 Metadata:
A@[0,2], C@[1,2]

Region 2 Metadata:
B@[2,4], C@[2,3]

Figure 3: Files are partitioned into multiple regions to decouple the size of metadata lists from the size of the file. This figure shows the fourth state of the file from Figure 2 partitioned into 2 MB regions. Writes that are entirely within a single region are appended solely to that region's metadata. Writes that cross regions are transactionally appended to multiple lists.

pointers to the metadata list. Readers retrieve the metadata list, compact it, and determine which slices must be retrieved from the storage servers to fulfill the read.

The correctness of this design relies upon the metadata storage providing primitives to atomically read and append to the list. HyperDex natively supports both of these operations. Because each writer writes slices before appending to the metadata list, it is guaranteed that any transaction that can see these immutable slices is serialized *after* the writing transaction commits. It can then retrieve the slices directly. The transactional guarantees of WTF extend directly from this design as well: a WTF transaction will execute a single HyperDex transaction consisting of multiple append and retrieve operations.

2.2 Storage Server Interface

The file slicing abstraction greatly simplifies the design of the storage servers. Storage servers deal exclusively with slices, and are oblivious to files, offsets, or concurrent writes. The minimal API required by file slicing consists of just two calls to create and retrieve slices.

A storage server processes a request to create a slice by writing the data to disk and returning a slice pointer to the caller. The structure of this request intentionally grants the storage server complete flexibility to store the slice anywhere it chooses because the slice pointer containing the slice's location is returned to the client only after the slice is written to disk. A storage server can retrieve slices by following the information in the slice pointer to open the named file, read the requisite number of bytes, and return them to the caller.

The direct nature of the slice pointer minimizes the bookkeeping required of the storage server implementation and permits a wide variety of implementation strategies. In the simplest strategy, which is the strategy used in the WTF implementation, each WTF storage server maintains a directory of slice-containing backing files and information about their own identities in the system. Each backing file is written sequentially as the storage server creates new slices.

As an optimization, each storage server maintains multiple backing files to which slices are appended. This serves three purposes: First, it allows servers to avoid contention when writing to the same file; second, it allows the storage server to spread data across multiple filesystems if configured to do so; and, finally, it allows the storage server to use hints provided by writers to improve locality on disk, as described in Section 2.7.

2.3 File Partitioning

Practically, it is desirable to keep the list of slice pointers small so that they can be stored, retrieved, and transmitted with low overhead; however, it would be impractical to achieve this by limiting the number of writes to a file. In order to achieve support for both arbitrarily large files and efficient operations on the list of slice pointers, WTF partitions a file into fixed size regions, each with its own list. Each region is stored as its own object in HyperDex under a deterministically derived key.

Operations on these partitioned metadata lists directly follow from the behavior of the system with a single metadata list. When an operation spans multiple regions, it is decomposed into one operation per region, and the decomposed operations execute within the context of a single HyperDex transaction. This guarantees that multi-region operations execute as one atomic action. Figure 3 shows a sample partitioning of a file, and how operations can span multiple metadata lists.

API	Description
`yank(fd,sz):slice,[data]`	Copy `sz` bytes from `fd`; return slice pointers and optionally the data
`paste(fd, slice)`	Write `slice` to `fd` and increment the offset
`punch(fd, amount)`	Zero-out `amount` bytes at the fd offset, freeing the underlying storage
`append(fd, slice)`	Append `slice` to the end of file `fd`
`concat(sources, dest)`	Concatenate the listed files to create dest
`copy(source, dest)`	Copy source to dest using only the metadata

Table 1: WTF's new file slicing API. Note that these supplement the POSIX API, which includes calls for moving a file descriptor's offset via `seek`. `concat` and `copy` are provided for convenience and may be implemented with `yank` and `paste`.

2.4 Filesystem Hierarchy

The WTF filesystem hierarchy is modeled after the traditional Unix filesystem, with directories and files. Each directory contains entries that are named links to other directories or files, and WTF enables files to be hard linked to multiple places in the filesystem hierarchy.

WTF implements a few changes to the traditional filesystem behavior to reduce the scope of a transaction when opening a file. Path traversal, as it is traditionally implemented, puts every directory along the path within the scope of a transaction, and requires multiple round trips to both HyperDex and the storage servers.

WTF avoids traversing the filesystem on open by maintaining a pathname to inode mapping. This enables a client to map a pathname to the corresponding inode with just one HyperDex lookup, no matter how deeply nested the pathname. To enable applications to enumerate the contents of a single directory, WTF maintains traditional-style directories, implemented as special files, alongside the one-lookup mapping. The two data structures are atomically updated using HyperDex transactions. This optimization simplifies the process of opening files, without significant loss of functionality.

Inodes are also stored in HyperDex, and contain standard information, such as link count and modification time. The inode also maintains ownership, group, and permissions information, though WTF differs from POSIX in that permissions are not checked on the full pathname from the root. Each inode also stores a reference to the highest-offset region for the file, enabling applications to find the end of the file. The inode refers to a region instead of a particular offset so that the inode is only written when the file grows beyond the bounds of a region, instead of every time the file changes in size.

Because HyperDex permits transactions to span multiple keys across independent schemas, updates to the filesystem hierarchy remain consistent. For example, to create a hardlink for a file, WTF atomically creates a new pathname to inode mapping for the file, increments the inode's link count, and inserts the pathname and inode pair into the destination directory, which requires a write to the file holding the directory entries.

2.5 File Slicing Interface

The file slicing interface enables new applications to make more efficient use of the filesystem. Instead of operating on bytes and offsets as traditional POSIX systems do, this new API allows applications to manipulate subsequences of files at the structural level, without copying or reading the data itself.

Table 1 summarizes the new APIs that WTF provides to applications. The `yank`, `paste`, and `append` calls are analogous to read, write, and append, but operate on slices instead of sequences of bytes. The `yank` call retrieves slice pointers for a range of the file. An application may provide these slice pointers to a subsequent call to `paste` or `append` to write the data back to the filesystem, reusing the existing slices. These write operations bypass the storage servers and only incur costs at the metadata storage component.

The `append` call is internally optimized to improve throughput. A naive `append` call could be implemented as a transaction that seeks to the end of the file, and performs a `paste`. While not incorrect, such an implementation would prohibit concurrency because only one append could commit for each value for the end of file. Instead, WTF stores alongside the metadata list an offset representing the end of the region. An `append` call translates to a conditional list append call within HyperDex that only succeeds when the current offset plus the length of the slice to be appended does not exceed the bounds of the region. When an append is too large to fit within a single region, WTF will fall back on reading the offset of the end of file, and performing a write at that offset. This enables multiple `append` operations to proceed in parallel in the common case.

The remaining calls in the file slicing API are provided for convenience, as they may be implemented in terms of `yank` and `paste`. `concat` concatenates multiple files to create one unified output file. `copy` copies a file by copying the file's compacted metadata.

2.6 Transaction Retry

To guarantee that WTF transactions never spuriously abort, WTF implements its own concurrency control that retries aborted metadata transactions. WTF operations in the client library often read metadata during the course of an operation that is not exposed to the calling application. A change to this data after it is read may force the metadata transaction to abort, but to abort the corresponding WTF transaction would be spurious from the perspective of the application.

For example, consider a file opened in "append" mode. Each write to the file must be written at the end-of-file offset, but the application does not learn this offset from the write. Internally, the client library computes the end of file, and then writes data at that offset. If the file changes in size between these two operations, the metadata transaction will abort. WTF masks this abort from the application by re-reading the end of file, and re-issuing the write at the new offset.

The mechanism that retries transactions is a thin layer between the WTF client library and the user's application. Each API call the application makes is logged in this layer by recording the arguments provided to the call and the value returned from the call. Should a metadata transaction abort during the WTF transaction commit, the WTF client library replays each operation from the log using the originally supplied arguments. If any replayed operation returns a value different from the logged call, the WTF transaction signals an abort to the application. Otherwise, WTF will commit the metadata changes from the replayed log to HyperDex. This process repeats as necessary until the metadata transaction, and, thus, the WTF transaction, commit, or a replayed operation triggers an WTF abort. This guarantees that WTF transactions are lockfree with zero spurious aborts.

To reduce the size of the replay log, the replay log refers to bytes of data that pass through the interface using slice pointers instead of copying the data. For example, a write of 100 MB will not be copied into the log; instead, the WTF client library writes the 100 MB to the requisite number of servers, and records the slice pointers in the log. Similarly, reads record slice pointers retrieved from the metadata, and not the slices themselves.

2.7 Locality-Aware Slice Placement

As an optimization, the WTF client library carefully places writes to the same region near each other on the storage servers to simultaneously improve locality for readers and to improve the efficiency of metadata compaction. When an application writes to a file sequentially, the locality-aware placement algorithm ensures that, with high probability, writes that appear consecutively in the metadata list will be consecutive on the storage servers' disks. During metadata compaction, the slice pointers for these consecutive writes are replaced by a single slice pointer that directly refers to the entire contiguous sequence of bytes on each storage server.

Two levels of consistent hashing [23] make it unlikely that two writes will map to the same backing files on the same storage server unless they are for the same metadata region. The WTF client library chooses the servers for each write by using consistent hashing across the list of storage servers. The client then provides the slice and identity of the metadata region to these servers, which use a different consistent hashing algorithm to map the write to disk. When collisions in the hash space do inevitably occur, it is unlikely that the colliding writes are issued so close in time as to be totally interleaved on disk in a way that eliminates opportunities for optimization.

2.8 Metadata Compaction and Defragmentation

The client library automatically compacts metadata during read and write operations to improve efficiency of future read and write operations. During write operations, the client library tracks the number of bytes written to both the metadata and the data for each region. When the ratio of metadata to data in a region exceeds a pre-defined threshold, the library retrieves the metadata list for the region, compacts it as shown in Figure 2, and writes the newly compacted list. When reading, the client compacts the metadata list via the same process.

When metadata compaction alone cannot reduce the metadata to data ratio below the pre-defined threshold, the client library defragments the list by rewriting the data. The library rewrites fragmented data within a region into one single slice and replaces the metadata list with a single pointer to this slice. For efficiency's sake, defragmentation happens only on read, not on writes,because the client library necessarily reads the fragmented slices to fulfill the read; it can rewrite the slices without the overall system paying the cost of reading the fragmented slices twice. This mechanism is unused in the common case because locality-aware slice placement avoids fragmentation.

2.9 Garbage Collection

WTF employs a garbage collection mechanism to prevent the number of unreferenced slices from growing without bound. Metadata compaction and defragmentation ensures that metadata will not grow without bound, but in the process creates garbage slices that are not referenced from anywhere in the filesystem.

Because WTF performs all bookkeeping within the metadata storage, storage servers cannot directly know which portions of its local data are garbage. One possible way to inform the storage servers would be to maintain a reference count for each slice. This method, however, would require that the reference count on the storage server be maintained within the scope of the metadata transactions. Doing so, while not infeasible, would significantly complicate WTF's design and require custom transaction handling on the storage servers.

Instead of reference counting, WTF periodically scans the entire filesystem metadata and constructs a list of in-use slice pointers for each storage server. For simplicity of implementation, these lists are stored in a reserved directory within the WTF filesystem so that they need not be maintained in memory or communicated out of band to the storage servers. Storage servers link the WTF client library and read the list of in-use slices to discover unused regions in their local storage space. The garbage collection mechanism runs periodically at a configurable interval that exceeds the longest-possible runtime of a transaction. Storage servers do not collect an unused slice until it appears in two or more consecutive scans.

Storage servers implement garbage collection by creating sparse files on the local disk. To compress a file containing garbage slices, a storage server rewrites the file, seeking past each unused slice. This creates a sparse file that occupies disk space proportional to the in-use slices it contains. Files with the most garbage are the most efficient to collect, because the garbage collection thread seeks past large regions of garbage and only writes the small number of remaining slices. Backing files with little garbage incur much more I/O, because there are more in-use slices to rewrite. WTF chooses the file with the most garbage to compact first, because it will simultaneously delete the most garbage and incur the least I/O. Some filesystems enable applications to selectively punch holes in the file without rewriting the data; although our implementation does not use these capabilities, an improved implementation could do so.

2.10 Fault Tolerance

WTF uses replication to add fault tolerance to the system. Changing WTF to be fault tolerant requires modifying the metadata lists' structure so that each entry references multiple replicas of the same data, each with a different slice pointer. On the write path, writers create multiple replica slices on distinct servers and append their pointers atomically as one list entry. Readers may read from any replica, as they hold identical data.

The metadata storage derives its fault tolerance from the guarantees offered by HyperDex. Specifically, that it can tolerate f concurrent failures for a user-configurable value of f. HyperDex uses value-dependent chaining to coordinate between the replicas and manage recovery from failures [16].

3 Implementation

Our implementation of WTF implements the file slicing abstraction. The implementation is approximately 30 k lines of code written. It relies upon HyperDex with transactions, which is approximately 85 k lines of code, with an additional 37 k lines of code of supporting libraries written for both projects. The replicated coordinator for both HyperDex and WTF is an additional 19 k lines of code. Altogether, WTF constitutes 171 k lines of code that were written for WTF or HyperDex.

WTF's fault tolerant coordinator maintains the list of storage servers and a pointer to the HyperDex cluster. It is implemented as a replicated object on top of Replicant, a Paxos-based replicated state machine service. The coordinator consists of just 960 lines of code that are compiled into a dynamically linked library that is passed to Replicant. Replicant deploys multiple copies of the library, and sequences function calls into the library.

4 Evaluation

To evaluate WTF, we will look at a series of both end-to-end and micro benchmarks that demonstrate WTF under a variety of conditions. The first part of this section looks at how the features of WTF may be used to implement a variety of end-to-end applications. We will then look at a series of microbenchmarks that characterize the performance of WTF's conventional filesystem interface.

All benchmarks execute on a cluster of fifteen dedicated servers. Each server is equipped with two Intel Xeon 2.5 GHz L5420 processors, 16 GB of DDR2 memory with ECC, and between 500 GB and 1 TB SATA spinning-disks. The servers are connected with gigabit ethernet via a single top of rack switch. Installed on each server is 64-bit Ubuntu 14.04, HDFS from Apache Hadoop 2.7, and WTF with HyperDex.

For all benchmarks, HDFS and WTF are configured similarly. Both systems are deployed with three nodes reserved for the meta-data—a single HDFS name node, or a HyperDex cluster—and the remaining twelve servers are allocated as storage nodes for the data. Clients are spread across the twelve storage nodes. Except for changes necessary to achieve feature parity, both systems were deployed in their default configuration. To bring the semantics of HDFS up to par with WTF, each `write` is followed by an `hflush` call to ensure that the write is flushed from the client-side buffer to HDFS. The `hflush` ensures that writes are visible to readers, and does *not* flush to disk. This is analogous to changing from the C library's `fwrite` to a UNIX `write` in a traditional application. The resulting guarantees are equivalent to those provided by WTF.

Additionally, in order to work around a bug with append operations [5], the HDFS block size was set to 64 MB. Without this change to the configuration, HDFS can report an out-of-disk-space condition when only 3% of the disk space is in use. Instead of gracefully handling the condition and falling back to other replicas as is done in WTF, the failure cascades and causes multiple writes to fail, making it impossible to complete some benchmarks. The change is unlikely to impact the performance of data nodes because the increase from 64 MB to

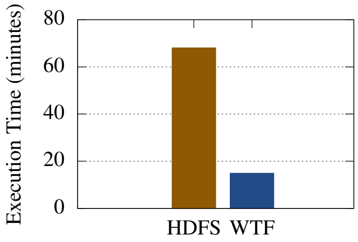


Figure 4: Total execution time for sorting 100 GB with map-reduce (512 kB rec.).

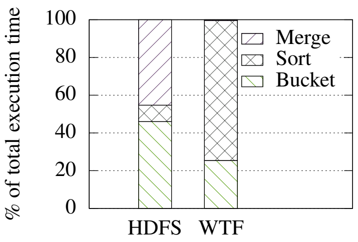


Figure 5: Execution time of sort broken down by stage of the map-reduce.

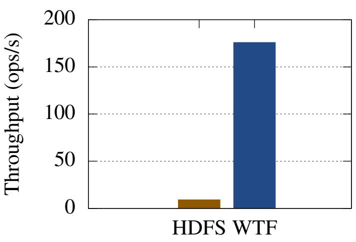


Figure 6: A concurrent work queue implementation.

128 MB was not motivated by performance [6]. WTF is also configured to use 64 MB regions.

Except where otherwise noted, both systems replicate all files such that two copies of the file exist. This allows the filesystem to tolerate the failure of any one storage server throughout the experiment without loss of data or availability. It is possible to tolerate more failures so long as all the replicas for a file do not fail simultaneously.

4.1 Applications

This section examines multiple applications that each demonstrate a different aspect of WTF's feature set.

Map Reduce: Sorting MapReduce [15] applications often build on top of filesystems like HDFS and GFS. In MapReduce, sorting a file is a three-step process that breaks the sort into two map jobs followed by a reduce job. The first map task partitions the input file into buckets, each of which holds a disjoint, contiguous section of the keyspace. These buckets are sorted in parallel by the second map task. Finally, the reduce phase concatenates the sorted buckets to produce the sorted output.

Each intermediate step of this application is written to the filesystem and the entire data set will be read or written several times over. Here, WTF's file slicing API can improve the efficiency of the application by reducing this excessive I/O. Instead of reading and writing whole records, WTF-based sort uses `yank` and `paste` to rearrange records. File slicing eliminates almost all I/O of the reduce phase using a `concat` operation.

Empirically, file slicing operations improve the running time of WTF-based sort. Figure 4 shows the total running time of both systems to sort a 100 GB file consisting of 500 kB records indexed by 10 B keys that were generated uniformly at random. In this benchmark, the intermediate files are written without replication because they may easily be recomputed from the input. We can see that WTF sorts the entire file in one fourth the time taken to perform the same task on HDFS.

The speedup is largely attributable file-slicing. From Figure 5, we can see that the WTF-based sorting application spends less time in the partitioning and merging steps than the HDFS-based sort. HDFS spends the majority of its execution time performing I/O tasks; just 8.5% of execution time is spent in the CPU-intensive sort task. In contrast, WTF spends 74.1% of its time in the CPU intensive task and seconds in the merge task.

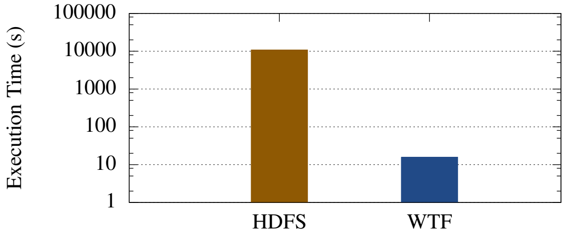


Figure 7: Time taken to generate a readily-playable video file from individual scenes.

Work Queue Work queues are a common component of large scale applications. Large work units may be durably written to the queue and handled by the application at a later point in time in FIFO order.

One simple implementation of a work queue is to use an append-only file as the queue itself. The application appends each work unit to the file, and can dequeue from the work queue by reading through the file sequentially—the file itself encodes the FIFO nature of the queue. This benchmark consists of an application with multiple writers that concurrently write to a single file on the filesystem. Each work unit is 1 MB in size and written atomically. The application runs on each client server, for a total of twelve application instances.

Figure 6 shows the aggregate throughput for the work queue built on top of both HDFS and WTF. We can see that WTF's throughput is 19× that of HDFS for this workload. Each work unit is saved to WTF in 55 ms, while the application built on HDFS waits 1.3 s on average to enqueue each work unit.

Image Host Image hosting sites, such as flickr or imgur have become the de-facto way of sharing images on the Internet. While imgur's implementation serves images from Amazon S3, Facebook's image serving solution, called Haystack [9], stores multiple photos in a single file to reduce the costs of reading and maintaining metadata. In Haystack, servers read into memory a map of the photos' locations on disk so that reading a single photo from disk does not entail any additional disk reads.

This example application models an imgur-like website built using the multi-photo file technique used within

Haystack. Photos are written to multi-gigabyte files, each of which has a footer mapping photos to their offsets in the files. The application loads this map into memory so that it may serve requests by locating the file's offset within the map, and then reading the file directly from the offset in the filesystem.

To better simulate a real photo-sharing website, photos are randomly generated to match the size of photos served by imgur. The distribution of photo sizes was collected from the front page of imgur.com over a 24-hour period. Because imgur serves both static images and gifs, the size of photos varies widely. Median image size is 332 kB, while the average image size is 8.5 MB. Because the precise request distribution of requests is not available from imgur, the workload re-uses the Zipf request distribution specified for YCSB workloads [13]. For this workload, we measured that WTF achieves 88.8% the throughput of the same application on top of HDFS. The performance difference is largely attributable to the reads of smaller files. As we will explore in the microbenchmarks section, WTF needs further optimization for small read and write operations.

Video Editing WTF's file slicing API can be used to reorganize large files with orders of magnitude less I/O. One particular domain where this can be useful is video editing of high-definition raw video. Such videos tend to be large in size, and will be rearranged frequently during the editing process. While specialized applications can edit and then play back videos, WTF enables another point in the design space.

This application uses WTF's file slicing to move scenes around in a video file without physically rewriting the video. The chief benefit of this design, over editors on existing filesystems, is that an off-the-shelf video player can play the edited video file because it is in a standard container format. To benchmark this application, we used our video editor to randomly rearrange the scenes in a 2 h movie, such that the movie out of chronological order. The source material was 1080p raw video dumped from a Bluray disk. Overall the raw video/audio occupies approximately 377 GB or 52 MB/s. Figure 7 shows the time taken to rewrite the file using HDFS's conventional API compared to WTF's file-slicing API. WTF takes three orders of magnitude less time to make a file readable—on the order of seconds—while conventional techniques require nearly three hours.

Sandboxing The transactional API of WTF makes it easy to use the filesystem as a sandbox where tasks may be committed or aborted depending on their outcome. The WTF implementation includes a FUSE module that enables users to mount the filesystem as if it were a local filesystem. This enables shell navigation of the filesystem hierarchy and allows regular applications to read and write WTF filesystems without modification. In addition to implementing the full filesystem interface, the FUSE bridge exposes special `ioctl`s to permit users to control transactions. Users may begin, abort, or commit transactions via command-line tools that wrap these `ioctl`s.

```
# wtf fuse ./mnt
# cd ./mnt
# wtf fuse-begin-transaction
# ls
/data.0000  /data.0001
/data.0002  /data.0003
....
# rm *
# ls
# wtf fuse-abort-transaction
# ls
/data.0000  /data.0001
/data.0002  /data.0003
....
```

Figure 8: WTF's transactional functionality enables users to manipulate the filesystem in isolation.

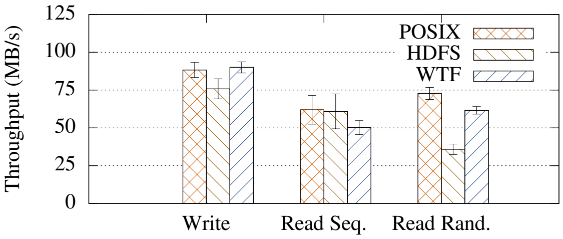


Figure 9: Performance of a one-server deployment of HDFS and WTF compared with the ext4 filesystem. Error bars indicate the standard error of the mean across seven trials.

The transactional features of the FUSE bridge enables users to perform risky actions within the context of a transaction; the transactional isolation provides a degree of safety users would otherwise not be afforded. The actions taken by the user are not visible until the user commits, and should the user abort, the actions will never be persisted to the filesystem. Figure 8 shows a sample interaction with an WTF filesystem containing data for a sample research project. We can see that the user begins a transaction and inadvertently removes all of the research data. Because the errant `rm` command happened in a transaction, the data remains untouched.

4.2 Micro Benchmarks

In this section we examine a series of microbenchmarks that quantify the performance of the POSIX API for both HDFS and WTF. Here HDFS serves as a gold-standard. With ten years of active development, and deployment across hundreds of nodes, including large deployments at both Facebook and LinkedIn [12], HDFS provides a reasonable estimate of distributed filesystem performance. Although we cannot expect WTF to grossly outperform HDFS—both systems are limited by the speed of the hard disks in the cluster—we can use the degree to which WTF and HDFS differ in performance to estimate the overheads present in WTF's design.

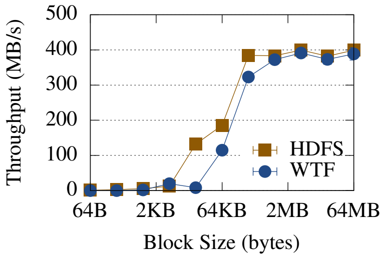


Figure 10: Throughput of a sequential write workload. Error bars report the standard error of the mean across seven trials[1].

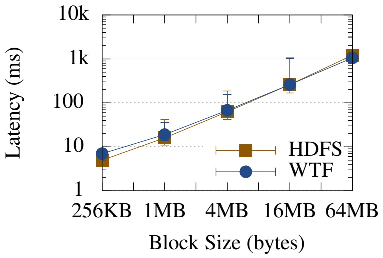


Figure 11: Median latency of write operations. Error bars report the 5th and 95th percentile latencies.

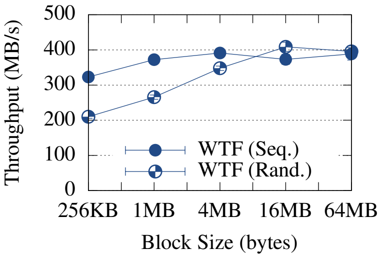


Figure 12: Throughput of a random write workload. Error bars report the standard error of the mean across seven trials.

Setup The workload for these benchmarks is generated by twelve distinct clients, one per storage server in the cluster, that all work in parallel. This configuration was chosen after experimentation because additional clients do not significantly increase the throughput, but do increase the latency significantly. All benchmarks operate on 100 GB of data, or over 16 GB per machine once replication is accounted for. This is large enough that our workload blocks on disk on Linux [25].

Single server performance This first benchmark executes on a single server to establish the baseline performance of a one node cluster. Here, we'll compare the two systems to each other and the same workload implemented on a local ext4 filesystem. The comparison to a local filesystem provides an upper bound on performance. To reduce the impact of round trip time in each distributed system the client and storage server are collocated. Figure 9 shows the throughput of write and read operations in the one-server cluster. From this we can see that the maximum throughput of a single node is 87 MB/s, which means the total throughput of the cluster peaks at approximately 1 GB/s.

Sequential Writes WTF guarantees that all readers in the filesystem see a write upon its completion. This benchmark examines the impact that write size has on the aggregate throughput achievable for filesystem-based applications. Figure 10 shows the results for block sizes between 64 B and 64 MB. For writes greater than 1 MB, WTF achieves 97% the throughput of HDFS. For 256 kB writes, WTF achieves 84% of the throughput of HDFS.

The latency for the two systems is similar, and directly correlated with the block size. Figure 11 shows the latency of writes across a variety of block sizes. We can see that WTF's median latency is very close to HDFS's median latency for larger writes, and that the 95th percentile latency for WTF is often lower than for HDFS.

Random Writes WTF enables applications to write at random offsets in a file without restriction. Because HDFS does not support random writes, we cannot use it as a baseline; instead, we will compare against the sequential write performance of WTF.

Figure 12 shows the aggregate throughput achieved by clients writing to random offsets within WTF files. We see that the random write throughput is always within a factor of two of the sequential throughput, and that throughput converges as the size of the writes approaches 8 MB.

Because the common case for a sequential write and a random write in WTF differ only at the stage where metadata is written to HyperDex, we expect that such a difference in throughput is directly attributable to the metadata stage. HyperDex provides lower latency variance to applications with a small working set than applications with a large working set with no locality of access. We can see the difference this makes in the tail latency of WTF writes in Figure 13, which shows the median and 99th percentile latencies for both the sequential and random workloads. The median latency for both workloads is the same for all block sizes. For block sizes 4 MB and larger, the 99th percentile latencies are approximately the same as well. Writes less than 4 MB in size exhibit a significant difference in 99th percentile latency between the sequential and random workloads. These smaller writes spend more time updating HyperDex than writing to storage servers. We expect that further optimization of HyperDex would close the gap between sequential and random write performance.

Sequential Reads Batch processing applications often read large input files sequentially during both the map and reduce phases. Although a properly-written application will double-buffer to avoid small reads, the filesystem should not rely on such behavior to enable high throughput. This experiment shows the extent to which WTF can be used by batch applications by reading through a file sequentially using a fixed-size buffer.

Figure 14 shows the aggregate throughput of concurrent readers reading through a 100 GB of data. We can see that for all read sizes, WTF's throughput is at least 80% the throughput of HDFS. The throughput reported here is double the throughput reported in the write benchmarks because only one of the two active replicas is consulted on each read. For smaller reads, WTF's throughput matches that of HDFS. The difference at larger sizes

[1] Blocks <256 kB wrote smaller files to limit execution time.

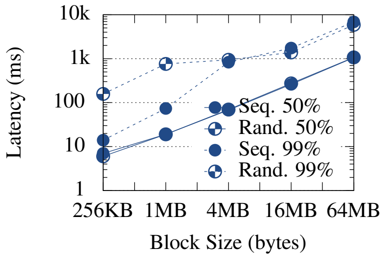


Figure 13: 50th/99th percentile latencies for sequential and random WTF writes.

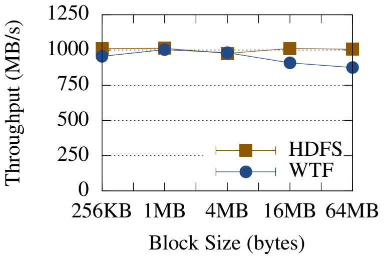


Figure 14: Throughput of a sequential read workload. Error bars report the standard error of the mean across seven trials.

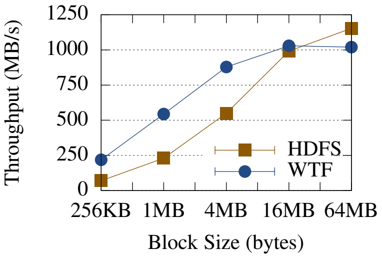


Figure 15: Throughput of a random read workload. Error bars indicate the standard error of the mean across seven trials.

is largely an artifact of the implementations. HDFS uses readahead on both the clients and storage servers in order to improve throughput for streaming workloads. By default, the HDFS readahead is configured to be 4 MB, which is the point at which the systems start to exhibit different characteristics. Our preliminary WTF implementation does not have any readahead mechanism, and exhibits lower throughput.

Random Reads Applications built on a distributed filesystem, such as key-value stores or record-oriented applications often require random access to the files. Figure 15 shows the aggregate throughput of twelve concurrent random readers reading from randomly chosen offsets within 100 GB of data. We can see that for reads of less than 16 MB, WTF achieves significantly higher throughput—at its peak, WTF's throughput is 2.4× the throughput of HDFS. Here, the readahead and client-side caching that helps HDFS with larger sequential read workloads adds overhead to HDFS that WTF does not incur. The 95th percentile latency of a WTF read is less than the median latency of a HDFS read for block sizes less than 4 MB.

Scaling Workload This experiment varies the number of clients writing to the filesystem to explore how concurrency affects both latency and throughput. This benchmark employs the workload from the sequential-write benchmark with a 4 MB write size and a variable number of workload-generating clients.

Figures 16 and 17 shows the resulting throughput and latency for between one and twelve clients. We can see that the single client performance is approximately 60 MB/s, while twelve clients sustain an aggregate throughput of approximately 380 MB/s. WTF's throughput is approximately the same as the throughput of HDFS for each data point. Running the same workload with forty-eight clients did not increase the throughput of either system beyond the throughput achieved with twelve clients, but did result in higher latency.

Fault Tolerance WTF's fault tolerance mechanism enables it to rapidly recover from failures. To demonstrate this mechanism, this benchmark performs sequential writes at a target throughput of 200 MB/s. Figure 18 shows the throughput of the benchmark over time. Thirty seconds into the benchmark, one storage server is taken offline; ten seconds later, the coordinator reconfigures the system to remove the failed storage server. In the time between the failure and reconfiguration, clients may try to use the failed server, fail to write to it, and fall back to another server. This increased effort is reflected in the lower throughput between failure and reconfiguration. After reconfiguration, throughput returns to to its rate before the failure. During the entire experiment, no writes failed, and the cluster as a whole remained available.

Garbage Collection This benchmark calculates the overhead of garbage collection on a storage server. As mentioned in Section 2.9, it is more efficient to collect files with more garbage than files with less garbage, and WTF preferentially garbage collects these larger files. Figure 19 shows the rate at which the cluster can collect garbage, for varying amounts of randomly located garbage, when all resources are dedicated to the task. We can see that when the cluster consists of 90% garbage, the cluster can reclaim this garbage at a rate of over 9 GB of garbage per second, because it need only write 1 GB/s to reclaim the garbage.

It is, however, impractical to dedicate all resources to garbage collection; instead, WTF dedicates only a fraction of I/O to the task. Storage servers initiate garbage collection when disk usage exceeds a configurable threshold, and ceases when the amount of garbage drops below 20%. Figure 19 shows that the maximum overhead required to maintain the system below this threshold is 4%.

Small Writes WTF's design is optimized for larger writes. The performance of smaller writes will largely be determined by the cost of updating the metadata. Writing a slice to the storage servers requires just one round trip because replicas are written to in parallel. Writing to the metadata store requires one round trip between client and the cluster, and multiple round trips within the cluster to propagate and commit the data. Further each write to the metdata requires writing approximately 50 B to HyperDex, so as writes to WTF shrink in size, the dominating

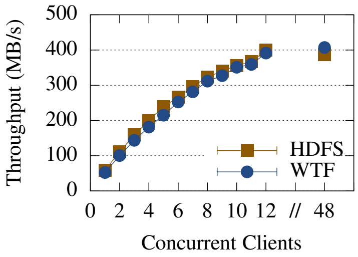


Figure 16: Throughput for varying numbers of writers. Error bars show the standard error of the mean across seven trials.

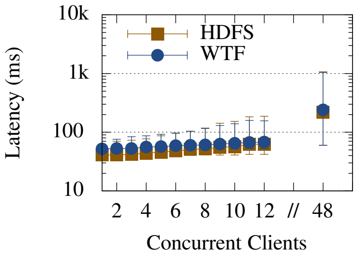


Figure 17: Median write latency for varying numbers of writers. Error bars show the 5th and 95th percentile latencies.

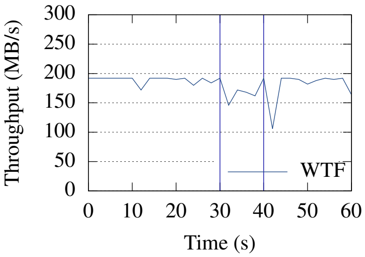


Figure 18: WTF tolerates failures—the failure occurs at the 30s mark—without a loss of availability.

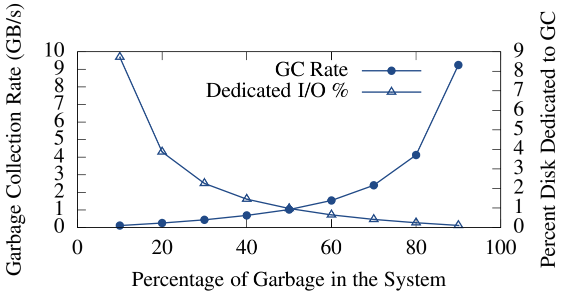


Figure 19: The maximum rate of garbage collection is positively correlated with the amount of garbage to be collected. Consequently, WTF dedicates a small fraction of its overall I/O to garbage collection.

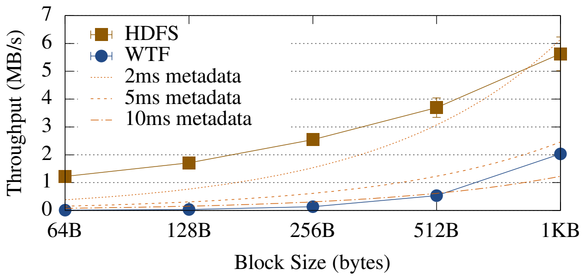


Figure 20: The time spent in metadata operations establishes an upper bound on the total throughput achievable by the system. This figure plots a portion of Figure 10 and theoretical maximum throughput for multiple metadata latencies.

cost becomes related to metadata.

Figure 20 focuses on a portion of the experiment shown in Figure 10, specifically writes less than 1 kB in size. HDFS achieves 140× higher throughput for 64 B writes, while the difference is only a factor of 2.8× for 1 kB writes. The figure also shows the calculated theoretical maximum throughput when the latency involved in writing to the metadata server is 2 ms, 5 ms, and 10 ms. This shows that the throughput of small operations is largely dependent upon the latency of metadata operations. Most workloads can avoid small operations with client side buffering, and further optimization of the metadata component could improve the throughput for small WTF writes.

5 Related Work

Filesystems have been an active research topic since the earliest days of systems research. Existing approaches related to WTF can be broadly classified into two categories based upon their design.

Distributed filesystems Distributed filesystems expose one or more units of storage over a network to clients. AFS [22] exports a uniform namespace to workstations, and stores all data on centralized servers. Other systems [21, 31, 33], most notably xFS [3] and Swift [10] stripe data across multiple servers for higher performance than can be achieved with a single disk. Petal [24] provides a virtual disk abstraction that clients may use as a traditional block device. Frangipani [38] builds a filesystem abstraction on top of Petal. NASD [20] and Panasas [42] employ customized storage devices that attach to the network to store the bulk of the metadata. In contrast to these systems, WTF provides transactional guarantees that can span hundreds or thousands of disks because its metadata storage scales independently of the number of storage servers.

Farsite [1] separates data from metadata to implement a byzantine fault tolerant filesystem where only the metadata replicas employ BFT algorithms. WTF uses a similar insight to leverage the transactional guarantees provided by the metadata storage to enable transactional guarantees to extend across the whole filesystem.

Recent work focuses on building large-scale datacenter-centric filesystems. GFS [19] and HDFS [4] employ a centralized master server that maintains the metadata, mediates client access, and coordinates the storage servers. Salus [41] improves HDFS to support storage and computation failures without loss of data, but retains the central metadata server. This centralized master approach, however, suffers from scalability bottlenecks inherent to the limits of a single server [27]. WTF overcomes the metadata scalability bottleneck using the scalable HyperDex key-value store [17].

CalvinFS [39] focuses on fast metadata management using distributed transactions in the Calvin [40] transaction processing system. Transactions in CalvinFS are limited, and cannot do read-modify-write operations on the filesystem without additional mechanism. Further, CalvinFS addresses file fragmentation using a heavy-weight garbage collection mechanism that entirely rewrites fragmented files; in the worst case, a sequential writer could incur I/O that scales quadratically in the size of the file. In contrast, WTF provides fully general transactions and carefully arranges data to improve sequential write performance.

Another approach to scalability is demonstrated by Flat Datacenter Storage [29], which enables applications to access any disk in a cluster via a CLOS network with full bisection bandwidth. To eliminate the scalability bottlenecks inherent to a single master design, FDS stores metadata on its tract servers and uses a centralized master solely to maintain the list of servers in the system. Blizzard [28] builds block storage, visible to applications as a standard block device, on top of FDS, using nested striping and eventual durability to service the smaller writes typical of POSIX applications. These systems are complementary to WTF, and could implement the storage server abstraction.

"Blob" storage systems behave similarly to file systems, but with a restricted interface that permits creating, retrieving, and deleting blobs, without efficient support for arbitrarily changing or resizing blobs. Facebook's f4 [37] ensures infrequently accessed files are readily available. Pelican [8] enables power-efficient cold storage by over provisioning storage, and selectively turning on subsets of disks to service requests. The design goals of these systems are different from the applications that WTF enables; WTF could be used in front of these systems to generate, maintain, and modify data before placing it into blob storage.

Transactional filesystems Transactional filesystems enable applications to offload much of the hard work relating to update consistency and durability to the filesystem. The QuickSilver operating system shows that transactions across the filesystem simplify application development [32]. Further work showed that transactions could be easily added to LFS, exploiting properties of the already-log-structured data to simplify the design [35]. Valor [36] builds transaction support into the Linux kernel by interposing a lock manager between the kernel's VFS calls and existing VFS implementations. In contrast to the transactions provided by WTF, and the underlying HyperDex transactions, these systems adopt traditional pessimistic locking techniques that hinder concurrency.

Optimistic concurrency control schemes often enable more concurrency for lightly-contended workloads. PerDiS FS adopts an optimistic concurrency control scheme that relies upon external components to reconcile concurrent changes to a file [18]. This allows users and applications to concurrently work on the same file. Liskov and Rodrigues show that much of the overhead of a serializable filesystem can be avoided by running read-only transactions in the recent past, and employing an optimistic protocol for read-write transactions [26]. WTF builds on top of HyperDex's optimistic concurrency and provides operations such as `append` that avoid creating conflicts between concurrent transactions.

WTF is not the first system to choose to employ a transactional datastore as part of its design. Inversion [30] builds on PostgreSQL to maintain a complete filesystem. KBDBFS [36] and Amino [43] both build on top of BerkeleyDB; the former is an in-kernel implementation of BerkeleyDB, while the latter eschews the complexity and takes a performance hit with a userspace implementation. WTF differs from these designs in that it stores solely the metadata in the transactional data store; data is stored elsewhere and not managed within the transactional component.

Stasis [34] makes the argument that no one design support all use cases, and that transactional components should be building blocks for applications. WTF's approach is similar: HyperDex's transactions are used as a base primitive for managing WTF's state, and WTF supports a transactional API. Applications built on WTF can use this API to achieve their own transactional behavior.

6 Conclusion

This paper described the Warp Transactional Filesystem (WTF), a new distributed filesystem that enables applications to operate on multiple files transactionally without requiring complex application logic. A new filesystem abstraction called *file slicing* further boosts performance by completely changing the filesystem interface to focus on metadata manipulation instead of data manipulation. Together, these features are a potent combination that enables a new class of high performance applications.

A broad evaluation shows that WTF achieves throughput and latency similar to industry-standard HDFS, while simultaneously offering stronger guarantees and a richer API. Sample applications show that WTF is usable in practice, and applications will often those built on a traditional filesystem—sometimes by orders of magnitude.

Acknowledgments

We'd like to thank our shepherd, Jon Howell, our anonymous reviewers, and Robbert van Renesse for their helpful feedback. We would like to thank Sean Ogden for his assistance in implementing WTF. This work is supported by the National Science Foundation under Grant No. CNS-1518779 and the Facebook Graduate Fellowship program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger Wattenhofer. FARSITE: Federated, Available, And Reliable Storage For An Incompletely Trusted Environment. In Proceedings of the *Symposium on Operating System Design and Implementation,* Boston, Massachusetts, December 2002.

[2] Amazon Web Services. Elastic Block Store. `http://aws.amazon.com/ebs/`.

[3] Thomas E. Anderson, Michael Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. In Proceedings of the *Symposium on Operating Systems Principles,* pages 109-126, Copper Mountain, Colorado, December 1995.

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

[5] Apache Hadoop Jira. DFS Used Space Is Not Correct Computed On Frequent Append Operations. `https://issues.apache.org/jira/browse/HDFS-6489`.

[6] Apache Hadoop Jira. Increase The Default Block Size. `https://issues.apache.org/jira/browse/HDFS-4053`.

[7] Apache HBase. `http://hbase.apache.org/`.

[8] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, David Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony I. T. Rowstron. Pelican: A Building Block For Exascale Cold Data Storage. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 351-365, Broomfield, Colorado, October 2014.

[9] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding A Needle In Haystack: Facebook's Photo Storage. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 47-60, Vancouver, Canada, October 2010.

[10] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using Distributed Disk Striping To Provide High I/O Data Rates. In *Computing Systems,* 4(4):405-436, 1991.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System For Structured Data. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 205-218, Seattle, Washington, November 2006.

[12] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John K. Ousterhout, and Mendel Rosenblum. Copysets: Reducing The Frequency Of Data Loss In Cloud Storage. In Proceedings of the *USENIX Annual Technical Conference,* San Jose, California, June 2013.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems With YCSB. In Proceedings of the *Symposium on Cloud Computing,* pages 143-154, Indianapolis, Indiana, June 2010.

[14] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 261-264, Hollywood, California, October 2012.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. In *Communications of the ACM,* 53(1):72-77, 2010.

[16] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-Value Store. In Proceedings of the *SIGCOMM Conference,* pages 25-36, Helsinki, Finland, August 2012.

[17] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Lightweight Multi-Key Transactions For Key-Value Stores. Cornell University, Ithaca, Technical Report, 2013.

[18] João Garcia, Paulo Ferreira, and Paulo Guedes. The PerDiS FS: A Transactional File System For A Distributed Persistent Store. In Proceedings of the *European SIGOPS Workshop,* pages 189-194, Sintra, Portugal, September 1998.

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In Proceedings of the *Symposium on Operating Systems Principles,* pages 29-43, Bolton Landing, New York, October 2003.

[20] Garth A. Gibson, David Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In Proceedings of the *Architectural Support for Programming Languages and Operating Systems,* pages 92-103, San Jose, California, October 1998.

[21] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. In *ACM Transactions on Computer Systems,* 13(3):274-310, 1995.

[22] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale And Performance In A Distributed File System. In *ACM Transactions on Computer Systems,* 6(1):51-81, 1988.

[23] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent Hashing And Random Trees: Distributed Caching Protocols For Relieving Hot Spots On The World Wide Web. In Proceedings of the *ACM Symposium on Theory of Computing,* pages 654-663, El Paso, Texas, May 1997.

[24] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In Proceedings of the *Architectural Support for Programming Languages and Operating Systems,* pages 84-92, Cambridge, Massachusetts, October 1996.

[25] Linux Kernel Developers. Documentation For /proc/sys/vm/*. `https://www.kernel.org/doc/Documentation/sysctl/vm.txt`.

[26] Barbara Liskov and Rodrigo Rodrigues. Transactional File Systems Can Be Fast. In Proceedings of the *European SIGOPS Workshop,* page 5, Leuven, Belgium, September 2004.

[27] Kirk McKusick and Sean Quinlan. GFS: Evolution On Fast-Forward. In *Communications of the ACM,* 53(3):42-49, 2010.

[28] James W. Mickens, Edmund B. Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, Cloud-Scale Block Storage For Cloud-Oblivious Applications. In Proceedings of the *Symposium on Networked System Design and Implementation,* pages 257-273, Seattle, Washington, April 2014.

[29] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen S. Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 1-15, Hollywood, California, October 2012.

[30] Michael A. Olson. The Design And Implementation Of The Inversion File System. In Proceedings of the *USENIX Winter Technical Conference,* pages 205-218, San Diego, California, January 1993.

[31] Frank B. Schmuck and Roger L. Haskin. GPFS: A Shared-Disk File System For Large Computing Clusters. In Proceedings of the *Conference on File and Storage Technologies,* pages 231-244, Monterey, California, January 2002.

[32] Frank B. Schmuck and James C. Wyllie. Experience With Transactions In QuickSilver. In Proceedings of the *Symposium on Operating Systems Principles,* pages 239-253, Pacific Grove, California, October 1991.

[33] Seagate Technology LLC. Lustre Filesystem. `http://lustre.org/`.

[34] Russell Sears and Eric A. Brewer. Stasis: Flexible Transactional Storage. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 29-44, Seattle, Washington, November 2006.

[35] Margo I. Seltzer. Transaction Support In A Log-Structured File System. In Proceedings of the *IEEE International Conference on Data Engineering,* pages 503-510, Vienna, Austria, April 1993.

[36] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling Transactional File Access Via Lightweight Kernel Extensions. In Proceedings of the *Conference on File and Storage Technologies,* pages 29-42, San Francisco, California, February 2009.

[37] Muralidhar Subramanian, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Sivakumar Viswanathan, Linpeng Tang, and Sanjeev Kumar. F4: Facebook's Warm BLOB Storage System. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 383-398, Broomfield, Colorado, October 2014.

[38] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In Proceedings of the *Symposium on Operating Systems Principles,* pages 224-237, Saint Malo, France, October 1997.

[39] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent WAN Replication And Scalable Metadata Management For Distributed File Systems. In Proceedings of the *Conference on File and Storage Technologies,* pages 1-14, Santa Clara, California, February 2015.

[40] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions For Partitioned Database Systems. In Proceedings of the *SIGMOD International Conference on Management of Data,* pages 1-12, Scottsdale, Arizona, May 2012.

[41] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness In The Salus Scalable Block Store. In Proceedings of the *Symposium on Networked System Design and Implementation,* pages 357-370, Lombard, Illinois, April 2013.

[42] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A. Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance Of The Panasas Parallel File System. In Proceedings of the *Conference on File and Storage Technologies,* pages 17-33, San Jose, California, February 2008.

[43] Charles P. Wright, Richard P. Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID Semantics To The File System. In *ACM Transactions on Storage,* 3(2), 2007.

BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores

Anurag Khandelwal
UC Berkeley

Rachit Agarwal
UC Berkeley

Ion Stoica
UC Berkeley

Abstract

We present BlowFish, a distributed data store that admits a smooth tradeoff between storage and performance for point queries. What makes BlowFish unique is its ability to navigate along this tradeoff curve efficiently at fine-grained time scales with low computational overhead.

Achieving a smooth and dynamic storage-performance tradeoff enables a wide range of applications. We apply BlowFish to several such applications from real-world production clusters: (i) as a data recovery mechanism during failures: in practice, BlowFish requires 5.4× lower bandwidth and 2.5× lower repair time compared to state-of-the-art erasure codes, while reducing the storage cost of replication from 3× to 1.9×; and (ii) data stores with spatially-skewed and time-varying workloads (*e.g.*, due to object popularity and/or transient failures): we show that navigating the storage-performance tradeoff achieves higher system-wide utility (*e.g.*, throughput) than selectively caching hot objects.

1 Introduction

Random access and *search* are the two fundamental operations performed on modern data stores. For instance, key-value stores [3, 5, 11, 15, 16, 18, 23, 25] and NoSQL stores [1, 4, 7, 12, 13, 17, 21, 30] support random access at the granularity of records. Many of these [1,4,7,17,21,22] also support search on records. These data stores typically store an amount of data that is larger than available fast storage[1], *e.g.*, SSD or main memory. The goal then is to maximize the performance using caching, that is, executing as many queries in faster storage as possible.

The precise techniques for efficiently utilizing cache vary from system to system. At a high-level, most data stores partition the data across multiple *shards* (partitions), with each server potentially storing multiple shards [1, 7, 21, 23]. Shards may be replicated and cached across multiple servers and the queries are load balanced across shard replicas [1, 4, 7, 12, 21].

To cache more shards, many systems use compression [1, 4, 7, 21]. Unfortunately, compression leads to a hard tradeoff between throughput and storage for the cached shards — when stored uncompressed, a shard can support high throughput but takes a larger fraction of available cache size; and, when compressed, takes smaller cache space but also supports lower throughput. Furthermore, switching between these two extreme points on the storage-performance tradeoff space cannot be done at fine-grained time scales since it requires compression or decompression of the entire shard. Such a hard storage-performance tradeoff severely limits the ability of existing data stores in many real-world scenarios when the underlying infrastructure [28,29], workload [9,10,14,26,31], or both changes over time. We discuss several such scenarios from real-world production clusters below (§1.1).

We present BlowFish, a distributed data store that enables a *smooth* storage-performance tradeoff between the two extremes (uncompressed, high throughput and compressed, low throughput), allowing fine-grained changes in storage and performance. What makes BlowFish unique is that applications can navigate from one operating point to another along this tradeoff curve *dynamically* over fine-grained time scales. We show that, in many cases, navigating this smooth tradeoff has higher system-wide utility (*e.g.*, throughput per unit of storage) than existing techniques. Intuitively, this is because BlowFish allows shards to increase/decrease the storage "fractionally", just enough to meet the performance goals.

1.1 Applications and summary of results

BlowFish, by enabling a dynamic and smooth storage-performance tradeoff, allows us to explore several problems from real-world production clusters from a different "lens". We apply BlowFish to three such problems:

Storage and bandwidth efficient data repair during failures. Existing techniques either require high storage (replication) or high bandwidth (erasure codes) for data repair, as shown in Table 1. By storing multiple replicas at different points on tradeoff curve, BlowFish can achieve the best of the two worlds — in practice, BlowFish requires storage close to erasure codes while requiring re-

[1] To support search, many of these systems store indexes in addition to the input, which further adds to the storage overhead. We collectively refer to the indexes combined with the input as "data".

Table 1: Storage and bandwidth requirements for erasure codes, replication and BlowFish for data repair during failures.

	Erasure (RS) Code	**Replication**	**BlowFish**
Storage	1.2×	3×	1.9×
Repair Bandwidth	10×	1×	1×

pair bandwidth close to replication. System state is restored by copying one of the replicas and navigating along the tradeoff curve. We explore the corresponding storage-bandwidth-throughput tradeoffs in §4.2.

Skewed workloads. Existing data stores can benefit significantly using compression [1, 4, 7, 12, 21]. However, these systems lose their performance advantages in case of dynamic workloads where (i) the set of hot objects changes rapidly over time [9, 14, 26, 31], and (ii) a single copy is not enough to efficiently serve a hot object. Studies from production clusters have shown that such workloads are a norm [9, 10, 14, 26, 31]. Selective caching [8], that caches additional replicas for hot objects, only provides coarse-grained support to handle dynamic workloads — each replica increases the throughput by 2× while incurring an additional storage overhead of 1×.

BlowFish not only provides a finer-grained tradeoff (increasing the storage overhead fractionally, just enough to meet the performance goals), but also achieves a better tradeoff between storage and throughput than selective caching of compressed objects. We show in §4.3 that BlowFish achieves 2.7–4.9× lower storage (for comparable throughput) and 1.5× higher throughput (for fixed storage) compared to selective caching.

Time-varying workloads. In some scenarios, production clusters delay additional replica creation to avoid unnecessary traffic (*e.g.*, for 15 minutes during transient failures [28, 29]). Such failures contribute to 90% of the failures [28, 29] and create high temporal load across remaining replicas. We show that BlowFish can adapt to such time-varying workloads even for spiked variations (as much as by 3×) by navigating along the storage-performance tradeoff in less than 5 minutes (§4.4).

1.2 BlowFish Techniques

BlowFish builds upon Succinct [7], a system that supports queries on compressed data[2]. At a high-level, Succinct stores two *sampled* arrays, whose sampling rate acts as a proxy for the compression factor in Succinct. BlowFish introduces *Layered Sampled Array* (LSA), a new data structure that stores sampled arrays using multiple layers of sampled values. Each combination of layers in LSA correspond to a static configuration of Succinct. Layers in LSA can be added or deleted transparently, independent of existing layers and query execution, thus enabling dynamic navigation along the tradeoff curve.

Each shard in BlowFish can operate on a different point on the storage-performance tradeoff curve. This leads to several interesting problems: how should shards (within and across servers) share the available cache? How should shard replicas share requests? BlowFish adopts techniques from scheduling theory, namely back-pressure style Join-the-shortest-queue [19] mechanism, to resolve these challenges in a unified and near-optimal manner. Shards maintain request queues that are used both to load balance queries as well as to manage shard sizes within and across servers.

In summary, this paper makes three contributions:

- Design and implementation of BlowFish, a distributed data store that enables a smooth storage-performance tradeoff, allowing fine-grained changes in storage and performance for each individual shard.
- Enables dynamic adaptation to changing workloads by navigating along the smooth tradeoff curve at fine-grained time scales.
- Uses techniques from scheduling theory to perform load balancing and shard management within and across servers.

2 BlowFish Overview

We briefly describe Succinct data structures in §2.1, with a focus on how BlowFish transforms these data structures to enable the desired storage-performance tradeoff. We then discuss the storage model and target workloads for BlowFish (§2.2). Finally, we provide a high-level overview of BlowFish design (§2.3).

2.1 Succinct Background

Succinct internally supports random access and search on flat unstructured files. Using a simple transformation from semi-structured data to unstructured data [7], Succinct supports queries on semi-structured data, that is, a collection of records. Similar to other key-value and NoSQL stores [1,3,4,12,15,21,23], each record has a unique identifier `key`, and a potentially multi-attribute `value`. Succinct supports random access via `get`, `put` and `delete` operations on keys; in addition, applications can `search` along individual attributes in values.

[2] Unlike Succinct, BlowFish does *not* enforce compression; some points on the tradeoff curve may have storage comparable to systems that store indexes along with input data.

Succinct supports random access and search using four data structures — Array-of-Suffixes (AoS), Input2AoS, AoS2Input and NextCharIdx (see Figure 1). AoS stores all suffixes in the input file in lexicographically sorted order. Input2AoS enables random access by mapping offsets in the input file to corresponding suffixes in the AoS. AoS2Input enables search by mapping suffixes in AoS to corresponding offsets in the input file. The Input2AoS and AoS2Input arrays do not possess any special structure, and require $n\lceil \log n \rceil$ space each for a file with n characters (since each entry is an integer in range 0 to $n-1$); Succinct reduces their space requirement using *sampling*. The fourth array, NextCharIdx, allows computing unsampled values in Input2AoS and AoS2Input. The AoS and the NextCharIdx arrays have certain structural properties that enable a compact representation. The description of AoS, NextCharIdx, and their compact representations is not required to keep the paper self-contained; we refer the reader to [7]. We provide necessary details on representation of Input2AoS and AoS2Input below.

Sampled Arrays: Storage versus Performance. Succinct reduces the space requirements of Input2AoS and AoS2Input using *sampling* — only a few sampled values (*e.g.*, for sampling rate α, value at indexes $0, \alpha, 2\alpha, ..$) from these two arrays are stored. NextCharIdx allows computing unsampled values during query execution.

The tradeoff is that for a sampling rate of α, the storage requirement for Input2AoS and AoS2Input is $2n\lceil \log n \rceil / \alpha$ and the number of operations required for computing each unsampled value is α.

Succinct thus has a fixed small storage cost for AoS and NextCharIdx, and the sampling rate α acts as a proxy for overall storage and performance in Succinct.

2.2 BlowFish data model and assumptions

BlowFish enables the same functionality as Succinct (§2.1) — support for random access and search queries on flat unstructured files, with extensions for key-value stores and NoSQL stores.

Assumptions. BlowFish makes two assumptions. First, *systems are limited by capacity of faster storage*, that is operate on data sizes that do not fit entirely into the fastest storage. Indeed, indexes to support search queries along with the input data makes it hard to fit the entire data in fastest storage especially for purely in-memory data stores (*e.g.*, Redis [5], MICA [23], RAMCloud [25]). Second, BlowFish assumes that data can be sharded in a manner that a query does not require touching each server in the system. Most real-world datasets and query workloads admit such sharding schemes [14, 26, 31].

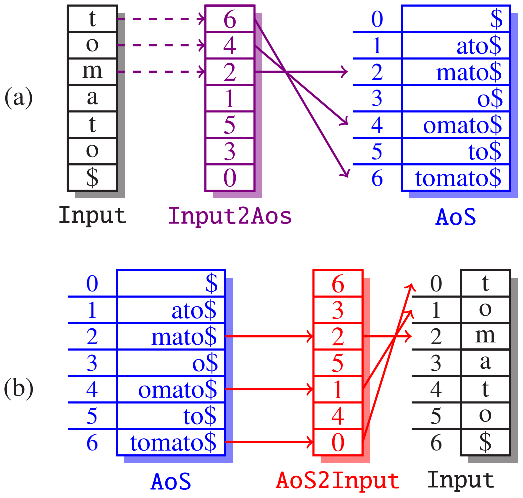


Figure 1: AoS stores suffixes in the input in lexicographically sorted order. (a) Input2AoS maps each index in the input to the index of the corresponding suffix in AoS. (b) Aos2Input maps each suffix index in AoS to the corresponding index in the input.

2.3 BlowFish Design Overview

BlowFish uses a system architecture similar to existing data stores, *e.g.*, Cassandra [21] and ElasticSearch [1]. Specifically, BlowFish comprises of a set of servers that store the data as well as execute queries (see Figure 2). Each server shares a similar design, comprising of multiple data shards (§3.1), a *request queue* per shard that keeps track of outstanding queries, and a special module *server handler* that triggers navigation along the storage-performance curve and schedules queries (§3.2).

Each shard admits the desired storage-performance tradeoff using *Layered Sampled Array* (LSA), a new data structure that allows transparently changing the sampling factor α for Input2AoS and AoS2Input over fine-grained time scales. Smaller values of α indicate higher storage requirements, but also lower latency (and vice versa). Layers can be added and deleted without affecting existing layers or query execution thus enabling dynamic navigation along the tradeoff curve. We describe LSA and the layer addition-deletion process in LSA in §3.1.

BlowFish allows each shard to operate at a different operating point on the storage-performance tradeoff curve (see Figure 3). Such a flexibility comes at the cost of increased dynamism and heterogeneity in system state. Shards on a server can have varying storage footprint and as a result, varying throughput. Moreover, storage footprint and throughput may vary across shard replicas. How should shards (within and across servers) share the available cache? How should shard replicas share requests? When should a shard trigger navigation along the storage-performance tradeoff curve?

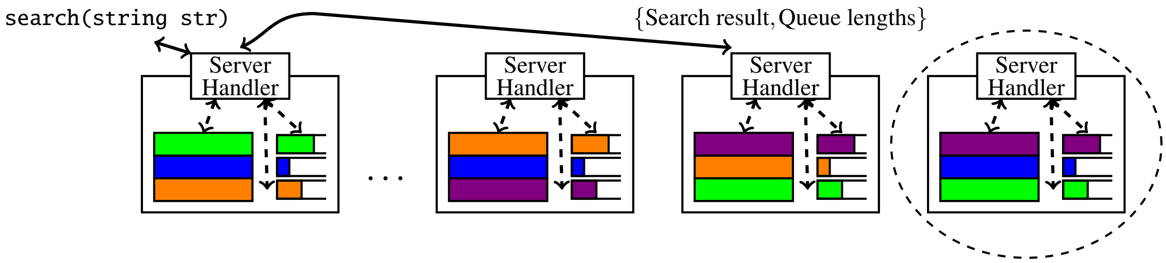


Figure 2: Overall BlowFish architecture. Each server has an architecture similar to the one shown in Figure 3. Queries are forwarded by Server Handlers to appropriate servers, and query responses encapsulate both results and queue lengths at that server.

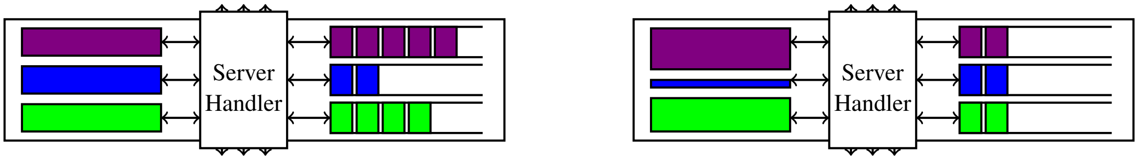


Figure 3: Main idea behind BlowFish: (left) the state of the system at some time t; (right) the state of the shards after BlowFish adapts — the shards that have longer outstanding queue lengths at time t adapt their storage footprint to a larger one, thus serving larger number of queries per second than at time t; the shards that have smaller outstanding queues, on the other hand, adapt their storage footprint to a smaller one thus matching the respective load.

BlowFish adopts techniques from scheduling theory, namely Join-the-shortest-queue [19] mechanism, to resolve the above questions in a unified manner. BlowFish servers maintain a *request queue* per shard, that stores outstanding requests for the respective shard. A server handler module periodically monitors request queues for local shards, maintains information about request queues across the system, schedules queries and triggers navigation along the storage-performance tradeoff curve.

Upon receiving a query from a client for a particular shard, the server handler forwards the query to the shard replica with shortest request queue length. All incoming queries are enqueued in the request queue for the respective shard. When the load on a particular shard is no more than its throughput at the current operating point on the storage-performance curve, the queue length remains minimal. On the other hand, when the load on the shard increases beyond the supported throughput, the request queue length for this shard increases (see Figure 3 (left)). Once the request queue length crosses a certain threshold, the navigation along the tradeoff curve is triggered either using the remaining storage on the server or by reducing the storage overhead of a relatively lower loaded shard. BlowFish internally implements a number of optimizations for selecting navigation triggers, maintaining request hysteresis to avoid unnecessary oscillations along the tradeoff curve, storage management during navigation and ensuring correctness in query execution during the navigation. We discuss these design details in §3.2.

3 BlowFish Design

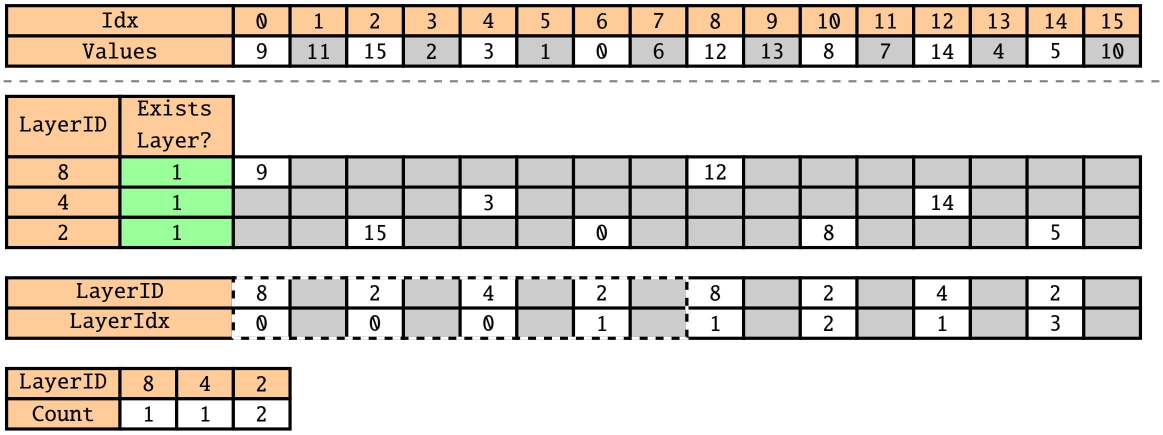
We start with the description of Layered Sampled Array (§3.1) and then discuss the system details (§3.2).

3.1 Layered Sampled Array

BlowFish enables a smooth storage-performance tradeoff using a new data structure, Layered Sampled Array (LSA), that allows dynamically changing the sampling factor in the two sampled arrays — Input2AoS and AoS2Input. We describe LSA below.

Consider an array `A`, and let `SA` be another array that stores a set of *sampled-by-index* values from `A`. That is, for *sampling rate* α, `SA[idx]` stores `A` value at index $\alpha \times$`idx`. For instance, if `A = {6, 4, 3, 8, 9, 2}`, the sampled-by-index array with sampling rate 4 and 2 are $\mathtt{SA}_4$ `= {6, 9}` and $\mathtt{SA}_2$ `= {6, 3, 9}`, respectively.

LSA emulates the functionality of SA, but stores the sampled values in multiple *layers*, together with a few auxiliary structures (Figure 4). Layers in LSA can be added or deleted transparently without affecting the existing layers. Addition of layers results in higher storage (lower sampling rate α) and lower query latency; layer deletion, on the other hand, reduces the storage but also increases the query latency. Furthermore, looking up a value in LSA is *agnostic* to the existing layers, independent of how many and which layers exist (pseudo code in Appendix A). This allows BlowFish to navigate along the storage-performance curve without any change in query execution semantics compared to Succinct.



Idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Values	9	11	15	2	3	1	0	6	12	13	8	7	14	4	5	10

LayerID	Exists Layer?																
8	1	9								12							
4	1					3								14			
2	1			15				0				8				5	

LayerID	8		2		4		2		8		2		4		2	
LayerIdx	0		0		0		1		1		2		1		3	

LayerID	8	4	2
Count	1	1	2

Figure 4: Illustration of *Layered Sampled Array* (LSA). The original unsampled array is shown above the dashed line (gray values indicate unsampled values). In LSA, each layer stores values for sampling rate given by LayerID, modulo values that are already stored in upper layers (in this example, sampling rates $8, 4, 2$). Layers are added and deleted at the bottom; that is, LayerID=2 will be added if and only if all layers with sampling rate $4, 8, 16, ..$ exist. Similarly, LayerID=2 will be the first layer to be deleted. The ExistsLayer bitmap indicates whether a particular layer exists (1) or not (0). LayerID and ExistsLayer allow checking whether or not value at any index idx is stored in LSA — we find the largest existing LayerID that is a proper divisor of idx. Note that among every consecutive 8 values in original array, 1 is stored in topmost layer, 1 in the next layer and 2 in the bottommost layer. This observation allows us to find the index into any layer LayerIdx where the corresponding sampled value is stored.

Layer Addition. The design of LSA as such allows arbitrary layers (in terms of sampling rates) to coexist; furthermore, layers can be added or deleted in arbitrary order. However, our implementation of LSA makes two simplifications. First, layers store sampled values for indexes that are *power of two*. Second, new layers are always added at the bottom. The rationale is that these two simplifications induce a certain structure in LSA, that makes the increase in storage footprint as well as time taken to add the layer very predictable. In particular, under the assumption that the unsampled array is of length $n = 2^k$ for some integer k, the number of sampled values stored at any layer is equal to the cumulative number of sampled values stored in upper layers (see Figure 4). If the sampling rate for the new layer is α, then this layer stores precisely $n/2\alpha$ sampled values; thus, the increase in storage becomes predictable. Moreover, since the upper layers constitute sampling rate 2α, computing each value in the new layer requires 2α operations (§2.1). Hence, adding a layer takes a fixed amount of time independent of the sampling rate of layer being added.

BlowFish supports two modes for creating new layers. In *dedicated layer construction*, the space is allocated for a new layer[3] and dedicated threads populate values in the layer; once all the values are populated the ExistsLayer bit is set to 1. The additional compute resources required in dedicated layer construction may be justified if the time spent in populating the new layer is smaller than the period of increased throughput experienced by the shard(s). However, such may not be the case for many scenarios.

The second mode for layer creation in BlowFish is *opportunistic layer construction*. This mode exploits the fact that the unsampled values for the two arrays are computed on the fly during query execution. A subset of the these values are the ones to be computed for populating the new layer. Hence, the query execution phase can be used to populate the new layer without using dedicated threads. The challenge in this mode is when to update the ExistsLayer flag — if set during the layer creation, the queries may incorrectly access values that have not yet been populated; on the other hand, the layer may remain unused if the flag is set after all the values are populated. BlowFish handles this situation by using a bitmap that stores a bit per sampled value for that layer. A set bit indicates that the value has already been populated and vice versa. The algorithm for opportunistic layer construction is outlined in Algorithm 2 in Appendix A.

It turns out that opportunistic layer construction performs really well for real-world workloads that typically follow a zipf-like distribution (repeated queries on certain objects). Indeed, the required unsampled values are computed during the first execution of a query and are thus available for all subsequent executions of the same query. Interestingly, this is akin to caching the query results without any explicit query result caching implementation.

[3] using free unused cache or by deleting layers from relatively lower loaded shards, as described in §3.2.4.

Layer Deletion. Deleting layers is relatively easier in BlowFish. To maintain consistency with layer additions, layer deletion proceeds from the bottom most layer. Layer deletions are computationally inexpensive, and do not require any special strategy. Upon the request for layer deletion, the `ExistsLayer` bitmap is updated to indicate that the corresponding layer is no longer available. Subsequent queries, thus, stop accessing the deleted layer. In order to maintain safety, we delay the memory deallocation for a short period of time after updating the `ExistsLayer` flag.

3.2 BlowFish Servers

We now provide details on the design and implementation of BlowFish servers.

3.2.1 Server Components

Each BlowFish server has three main components (see Figure 2 and Figure 3):

Data shards. Each server stores multiple data shards, typically one per CPU core. Each shard stores the two sampled arrays — Input2AoS and AoS2Input — using LSA, along with other data structures in Succinct. This enables a smooth storage-performance tradeoff, as described in §3.1. The aggregate storage overhead of the shards may be larger than available main memory. Each shard is memory mapped; thus, only the most accessed shards may be paged into main memory.

Request Queues. BlowFish servers maintain a queue of outstanding queries per shard, referred to as *request queues*. The length of request queues provide a rough approximation to the load on the shard — larger request queue lengths indicate a larger number of outstanding requests for the shard, implying that the shard is observing more queries than it is able to serve (and vice versa).

Server Handler. Each server in BlowFish has a server handler module that acts as an interface to clients as well as other server handlers in the system. Each client connects to one of the server handlers that handles the client query (similar to Cassandra [21]). The server handler interacts with other server handlers to execute queries and to maintain the necessary system state. BlowFish server handlers are also responsible for query scheduling and load balancing, and for making decisions on how shards share the cache available at the *local* server. We discuss these functionalities below.

3.2.2 Query execution

Similar to existing data stores [1, 4, 21], an incoming query in BlowFish may touch one or more shards depending on the sharding scheme. The server handler handling the query is responsible for forwarding the query to the server handler(s) of the corresponding shard(s); we discuss query scheduling across shard replicas below. Whenever possible, the query results from multiple shards on the same server are aggregated by the server handler.

Random access and search. BlowFish does *not* require changes in Succinct algorithms for executing queries at each shard, with the exception of looking up values in sampled arrays[4]. In particular, since the two sampled arrays in Succinct — Input2AoS and AoS2Input — are replaced by LSA, the corresponding lookup algorithms are replaced by lookup algorithms for LSA (§2.3, Figure 4). We note that, by using `ExistsLayer` flag, BlowFish makes LSA lookup algorithms transparent to existing layers and query execution.

Updates. BlowFish implements data appends exactly as Succinct [7] does. Specifically, BlowFish uses a multistore architecture with a write-optimized LogStore that supports fine-grained appends, a query-optimized SuffixStore that supports bulk appends and a memory-optimized SuccinctStore. LogStore and SuffixStore, for typical cluster configurations, store less than 0.1% of the entire dataset (the most recently added data). BlowFish does not require changes in LogStore and SuffixStore implementation, and enables the storage-performance tradeoff for data only in SuccinctStore. Since the storage and the performance of the system is dominated by SuccinctStore, the storage-performance tradeoff curve of BlowFish is not impacted by update operations.

3.2.3 Scheduling and Load Balancing

BlowFish server handlers maintain the request queue lengths for each shard in the system. Each server handler periodically monitors and records the request queue lengths for *local* shards. For non-local shards, the request queue lengths are collected during the query phase — server handlers encapsulate the request queue lengths for their local shards in the query responses. Upon receiving a query response, a server handler decapsulates the request queue lengths and updates its local metadata to record the new lengths for the corresponding shards.

Each shard (and shard replica) in BlowFish may operate on a different point on the storage-performance curve (Figure 3). Thus, different replicas of the same shard may have different query execution time for the same query. To efficiently schedule queries across such a heterogeneous system, BlowFish adopts techniques from scheduling theory literature — a back-pressure scheduling style Join-the-shortest-queue [19] mechanism. An incoming query

[4] The description of these algorithms is not required to keep the paper self-contained; we refer the reader to [7] for details.

for a shard is forwarded to the replica with the smallest request queue length. By conceptually modeling this problem as replicas having the same speed but varying job sizes (for the same query), the analysis for Join-the-shortest-queue [19] applies to BlowFish, implying close to optimal load balancing.

3.2.4 Dynamically Navigating the Tradeoff

BlowFish uses the request queues not only for scheduling and load balancing, but also to trigger navigation along the storage-performance tradeoff curve for each individual shard. We discuss below the details on tradeoff navigation, and how this enables efficient cache sharing among shards within and across servers.

One challenge in using request queue lengths as an approximation to load on the shard is to differentiate short-term spikes from persistent overloading of shards (Figure 5). To achieve this, BlowFish server handlers also maintain exponentially averaged queue lengths for each local shard — the queue lengths are monitored every δ time units, and the exponentially averaged queue length at time t is computed as:

$$Q_t^{avg} = \beta \times Q_t + (1-\beta) \times Q_{t-\delta}^{avg} \qquad (1)$$

The parameters β and δ provide two knobs for approximating the load on a shard based on its request queue length. β is a fraction ($\beta < 1$) that determines the contribution of more recent queue length values to the average — larger β assigns higher weight to more recent values in the average. δ is the periodicity at which queue lengths are averaged — smaller values of δ (i.e., more frequent averaging) results in higher sensitivity to bursts in queue length. Note that a small exponentially average queue length implies a persistently underloaded shard.

We now describe how shards share the available cache within and across servers by dynamically navigating along the storage-performance tradeoff curve. We start with the relatively simpler case of shards on the same server, and then describe the case of shards across servers.

Shards on the same server. Recall that BlowFish implementation adds and deletes layers in a bottom-up fashion, with each layer storing sampled values for powers of two. Thus, at any instant, the sampling rate of LSA is a power of two $(2, 4, 8, \ldots)$. For each of these sampling rates, BlowFish stores two *threshold* values. The *upper threshold* value is used to trigger storage increase for any particular shard — when the exponentially averaged queue length of a shard S crosses the upper threshold value, S must be consistently overloaded and must increase its throughput.

However, the server may not have extra cache to sustain the increased storage for S. For such scenarios, BlowFish stores a *lower threshold* value which is used to trigger storage reduction. In particular, if the exponentially averaged queue length *and* the instantaneous request queue length for one of the other shards S' on the same server is below the lower threshold, BlowFish reduces the storage for S' before triggering the storage increase for S. If there is no such S', the server must already be throughput bottlenecked and the navigation for S is not triggered.

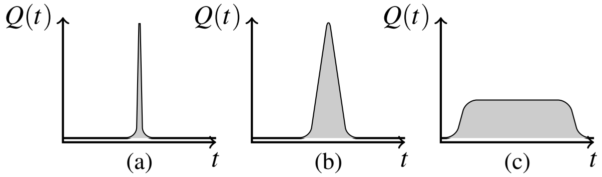


Figure 5: Three different scenarios of queue length ($Q(t)$) variation with time (t). (a) shows a very short-lasting "spike", (b) shows a longer lasting spike while (c) shows a persistent "plateau" in queue-length values. BlowFish should ideally ignore spikes as in (a) and attempt to adapt to the queue length variations depicted in (b) and (c).

We make two observations. First, the goals of exponentially averaged queue lengths and two threshold values are rather different: the former makes BlowFish stable against temporary spikes in load, while the latter against "flap damping" of load on the shards. Second, under stable loads, the above technique for triggering navigation along the tradeoff curve allows each shard on the same server to share cache proportional to its throughput requirements.

Shard replicas across servers. At the outset, it may seem like shards (and shard replicas) across servers need to coordinate among themselves to efficiently share the total system cache. It turns out that local cache sharing, as described above, combined with BlowFish's scheduling technique implicitly provides such a coordination.

Consider a shard S with two replicas R1 and R2, both operating at the same point on the tradeoff curve and having equal queue lengths. The incoming queries are thus equally distributed across R1 and R2. If the load on S increases gradually, both R1 and R2 will eventually experience load higher than the throughput they can support. At this point, the request queue lengths at R1 and R2 start building up at the same rate. Suppose R2 shares the server with other heavily loaded shards (that is, R2 can not navigate up the tradeoff curve). BlowFish will then trigger a layer creation for R1 only. R1 can thus support higher throughput and its request queue length will decrease. BlowFish's scheduling technique kicks in here: incoming queries will now be routed to R1 rather than equal load balancing, resulting in lower load at R2. It is easy to see that at this point, BlowFish will load balance queries to R1 and R2 proportional to their respective throughputs.

4 Evaluation

BlowFish is implemented in ≈ 2K lines of C++ on top of Succinct [7]. We apply BlowFish to application domains outlined in §1.1 and compare its performance against state-of-the-art schemes for each application domain.

Evaluation Setup. We describe the setup used for each application in respective subsections. We describe here what is consistent across all the applications: dataset and query workload. We use the TPC-H benchmark dataset [6], that consists of records with 8 byte keys and roughly 140 byte values on an average; the values comprise of 15 attributes (or columns). We note that several of our evaluation results are independent of the underlying dataset (*e.g.*, bandwidth for data repair, time taken to navigate along the tradeoff curve, etc.) and depend only on amount of data per server.

We use a query workload that comprises of 50% random access queries and 50% search queries; we discuss the impact of varying the fraction of random access and search queries in §4.1. Random access queries return the entire value, given a key. Search queries take in an (attribute, value) pair and return all keys whose entry for the input attribute matches the value. We use three query distributions in our evaluation for generating queries over the key space (for random access) and over the attribute values (for search). First, *uniform distribution* with queries distributed uniformly across key space and attribute values; this essentially constitutes a worst-case scenario for BlowFish[5]. The remaining two query workloads follow *Zipf distribution with skewness* 0.99 *(low skew) and* 0.01 *(heavily skewed)*, the last one constituting the best-case scenario for BlowFish.

All our distributed experiments run on Amazon EC2 cluster comprising of c3.2xlarge servers, with 15GB RAM backed by two 80GB SSDs and 8 vCPUs. Unless mentioned otherwise, all our experiments shard the input data into 8GB shards and use one shard per CPU core.

4.1 Storage Performance Tradeoff

We start by evaluating the storage-performance tradeoff curve enabled by BlowFish. Figure 6 shows this tradeoff for query workload comprising of 50% random access and 50% search queries; Appendix B presents the curves for other workloads. Note that the tradeoff for mixed workload has characteristics similar to 100% `search` workload (Appendix B) since, similar to other systems, execution time for search is significantly higher than random access. The throughput is, thus, dominated by search latency.

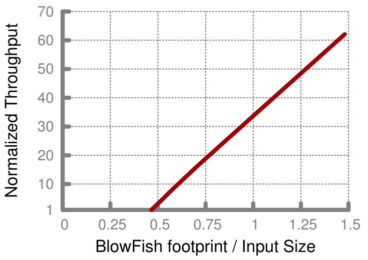


Figure 6: Storage-throughput tradeoff curve (per thread) enabled by BlowFish. The y-axis is normalized by the throughput of smallest possible storage footprint (71ops) in BlowFish.

We make two observations in Figure 6. First, BlowFish achieves storage footprint varying from 0.5× to 8.7× the input data size (while supporting search functionality; the figure shows only up to 1.5× the data size for clarity)[6]. In particular, BlowFish does not enforce compression. Second, increase in storage leads to super-linear increase in throughput (moving from ≈ 0.5 to ≈ 0.75 leads to 20× increase in throughput) due to non-linear computational cost of operating on compressed data [7].

4.2 Data Repair During Failures

We now apply BlowFish to the first application: efficient data recovery upon failures.

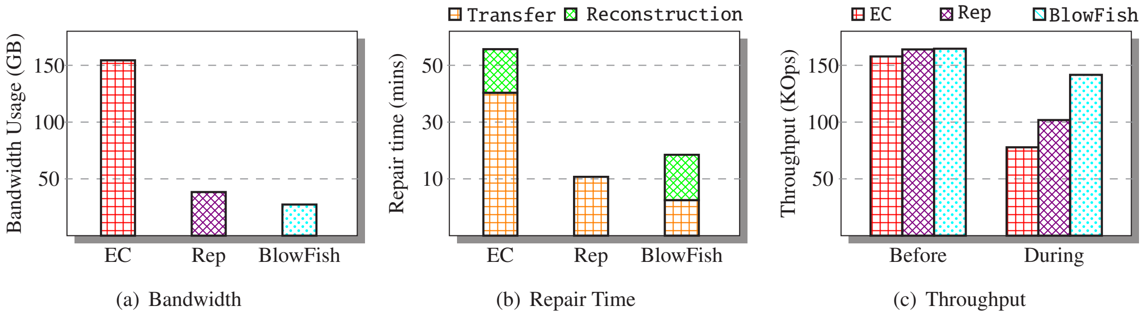
Existing techniques and BlowFish tradeoffs. Two techniques exist for data repair during failures: replication and erasure codes. The main tradeoff is that of storage and bandwidth, as shown in Table 1. Note that this tradeoff is hard; that is, for both replication and erasure codes, the storage overhead and the bandwidth for data repair is fixed for a fixed fault tolerance. We discuss related work in §5, but note that erasure codes remain inefficient for data stores serving small objects due to high repair time and/or bandwidth requirements.

4.2.1 Experimental Setup

We perform evaluation along four metrics: storage overhead, bandwidth and time required for data repair, and throughput before and during failures. Since none of the open-source data stores support erasure codes, we use an implementation of Reed-Solomon (RS) codes [2]. The code use 10 data blocks and 2 parity blocks, similar to those used at Facebook [24, 29], but for two failure case. Accordingly, we use 3× replication. For BlowFish, we use an instantiation that uses three replicas with storage $0.9\times, 0.5\times$ and 0.5×, aggregating to 1.9× storage — an operating point between erasure codes and replication.

[5] Intuitively, queries distributed uniformly across shards and across records alleviates the need for shards having varying storage footprints.

[6] The smallest footprint is 0.5× since TPC-H data is not very compressible, achieving compression factor of 3.1 using gzip.



(a) Bandwidth

(b) Repair Time

(c) Throughput

Figure 7: Comparison of BlowFish against RS erasure codes and replication (discussion in §4.2.2). BlowFish requires 5.4× lower bandwidth for data repair compared to erasure codes, leading to 2.5× faster repair time. BlowFish achieves throughput comparable to erasure codes and replication under no failures, and 1.4 − 1.8× higher throughput during failures.

We use 12 server EC2 cluster to put data and parity blocks on separate servers; each server contains both data and parity blocks, but not for the same data. Replicas for replication and BlowFish were also distributed similarly. We use 160GB of total raw data distributed across 20 shards. The corresponding storage for erasure codes, replication and BlowFish is, thus, 192, 480 and 310GB. Note that the cluster has 180GB main memory. Thus, all data shards for erasure codes fit in memory, while a part of BlowFish and replication data is spilled to disk (modeling storage-constrained systems).

We use uniform query distribution (across shards and across records) for throughput results. Recall that this distribution constitutes a worst-case scenario for BlowFish. We measure the throughput for the mixed 50% random access and 50% search workload.

4.2.2 Results

Storage and Bandwidth. As discussed above, RS codes, replication and BlowFish have a storage overhead of 1.2×, 3× and 1.9×. In terms of bandwidth, we note that the three schemes require storing 16, 40 and 26GB of data per server, respectively. Figure 7(a) shows the corresponding bandwidth requirements for data repair for the three schemes. Note that while erasure codes require 10× bandwidth compared to replication *for each individual failed shard*, the overall bandwidth requirements are less than 10× since each server in erasure coded case also stores lesser data due to lower storage footprint of erasure codes (best case scenario for erasure codes along all metrics).

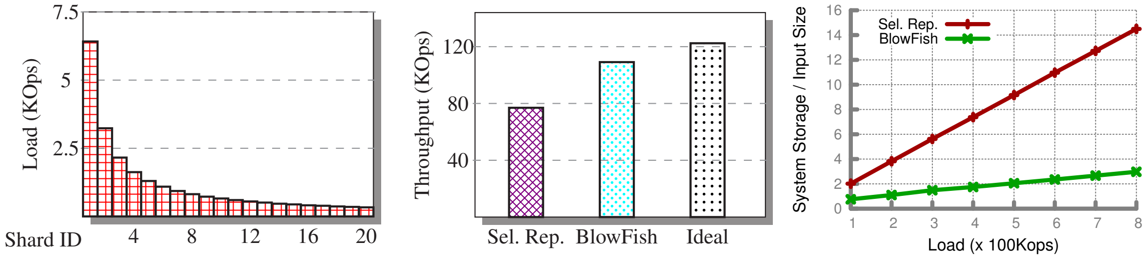
Repair time. The time taken to repair the failed data is a sum of two factors — time taken to copy the data required for recovery (transfer time), and computations required by the respective schemes to restore the failed data (reconstruction time). Figure 7(b) compares the data repair time for BlowFish against replication and RS codes.

RS codes require roughly 5× higher transfer time compared to BlowFish. Although erasure codes read the required data in parallel from multiple servers, the access link at the server where the data is being collected becomes the network bottleneck. This is further exacerbated since these servers are also serving queries. The decoding time of RS codes is similar to reconstruction time for BlowFish. Overall, BlowFish is roughly 2.5× faster than RS codes and 1.4× slower than replication in terms of time taken to restore system state after failures.

Throughput. The throughput results for the three schemes expose an interesting tradeoff (see Figure 7(c)).

When there are no failures, all the three schemes achieve comparable throughput. This is rather non-intuitive since replication has three replicas to serve queries while erasure codes have only one and BlowFish has replicas operating at smaller storage footprints. However, recall that the cluster is bottlenecked by the capacity of faster storage. If we load balance the queries in replication and in BlowFish across the three replicas, many of these queries are executed off SSD, thus reducing the overall system throughput (much more for replication since many more queries are executed off SSD). To that end, we evaluated the case of replication and BlowFish where queries are load balanced to only one replica; in this case, as expected, all the three schemes achieve comparable throughput.

During failures, the throughput for both erasure codes and replication reduces significantly. For RS codes, 10 out of (remaining) 11 servers are used to both read the data required for recovery as well as to serve queries. This severely affects the overall RS throughput (reducing it by 2×). For replication, note that the amount of failed data is 40GB (five shards). Recovering these shards results in replication creating two kinds of interference: interfering



(a) Load distribution across shards

(b) Throughput for a fixed storage

(c) Storage required to sustain load

Figure 8: Comparison of BlowFish and selective caching for skewed workload application. See §4.3 for discussion.

with queries being answered on data unaffected by failures *and* queries answered on failed server now being answered off-SSD from remaining servers. This interference reduces the replication throughput by almost 33%. Note that both these interferences are minimal in BlowFish: fewer shards need be constructed, thus fewer servers are interfered with, and fewer queries go to SSD. It turns out that the interference is minimal, and BlowFish observes minimal throughput reduction (less than 12%) during failures. As a result, BlowFish throughput during failures is is $1.4-1.8\times$ higher than the other two schemes.

4.3 Skewed Workloads

We now apply BlowFish to the problem of efficiently utilizing the system cache for workloads with skewed query distribution across shards (*e.g.*, more queries on hot data and fewer queries on warm data). The case of skew across shards varying with time is evaluated in next subsection.

State-of-the-art. The state-of-the-art technique for handling spatially-skewed workloads in Selective caching [8] that caches, for each object, number of replicas proportional to the load on the object.

4.3.1 Experimental Setup

We use 20 data shards, each comprising of 8GB of raw data, for this experiment. We compare BlowFish and Selective caching using two approaches. In the first approach, we fix the cluster (amount of fast storage) and *measure* the maximum possible throughput that each scheme can sustain. In the second approach, we vary the load for the two schemes and *compute* the amount of fast storage required by each scheme to sustain that load.

For the former, we use a cluster with 8 EC2 servers. A large number of clients generate queries with a Zipf distribution with skewness 0.01 (heavily skewed) across the shards. As shown in Figure 8(a), the load on the heaviest shard using this distribution is 20× the load on the lightest shard — this models the real-world scenario of a few shards being "hot" and most of the shards being "cold". For selective caching, each shard has number of replicas proportional to its load (recall, total storage is fixed); for BlowFish, the shard operates at a point on the tradeoff curve that can sustain the load with minimal storage overhead. We distribute the shards randomly across the available servers. For the latter, we vary the load and compute the amount of fast storage required by the two schemes to meet the load assuming that the entire data fits in fast storage. Here, we increase the number of shards to 100 to perform computations for a more realistic cluster size.

4.3.2 Results

For fixed storage. The storage required for selective caching and BlowFish to meet the load is 155.52GB and 118.96GB, respectively. Since storage is constrained, some shards in selective caching can not serve queries from faster storage. Intuitively, this is because BlowFish provides a finer-grained tradeoff (increasing the storage overhead fractionally, just enough to meet the performance goals) compared to the coarse-grained tradeoff of selective replication (throughput can be increased only by 2× by adding another replica requiring 1× higher storage overhead). Thus, BlowFish utilizes the available system cache more efficiently. Figure 8(b) shows that this leads to BlowFish achieving 1.5× higher throughput than selective caching. Interestingly, BlowFish achieves 89% of the ideal throughput, where the ideal is computed by taking into account the load skew across shards, the total system storage, the maximum possible per-shard throughput per server, and by placing heavily loaded shards with lightly loaded shards. The remaining 11% is attributed to the random placement of shards across servers, resulting in some servers being throughput bottlenecked.

Fixed load. Figure 8(c) shows that, as expected, BlowFish requires $2.7-4.9\times$ lower amount of fast storage compared to selective caching to sustain the load.

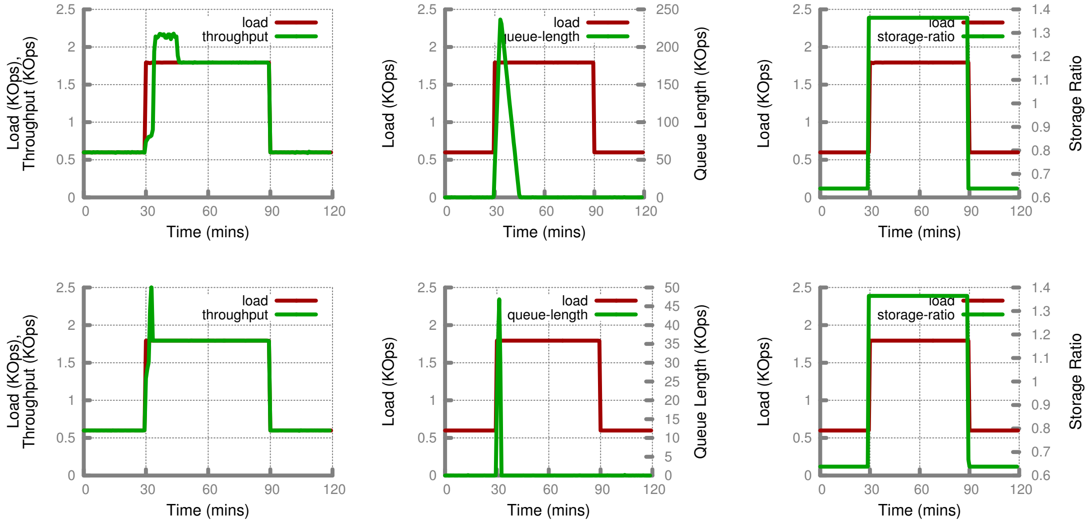


Figure 9: Opportunistic layer construction with spiked changes in load for uniform workload (top three) and skewed workload (bottom three). The figures show variation in throughput (left), request queue length (center) and storage footprint (right).

4.4 Time-varying workloads

We now evaluate BlowFish's ability to *adapt* to time-varying load, in terms of time taken to adapt and queue stability. We also evaluate the performance of BlowFish's scheduling technique during such time-varying loads.

4.4.1 Experimental Setup

We perform micro-benchmarks to focus on adaptation time, queue stability and per-thread shard throughput for time-varying workloads. We use a number of clients to generate time-varying load on the system. We performed four sets of experiments: uniform and skewed (Zipf with skewness 0.01) query distribution (across queried keys and search terms); and, gradual and spiked variations in load. It is easy to see that (uniform, spiked) and (skewed, gradual) are the worst-case and the best-case scenario for BlowFish, respectively. We present results for spiked variations in load (*e.g.*, due to transient failures) for both uniform and skewed query distribution; the remaining results are in Appendix C. We perform micro-benchmarks by increasing the load on the shard from 600ops to 1800ops suddenly (3× increase in load models failures of two replicas, an extremely unlikely scenario) at time $t = 30$ and observe the system for an hour before dropping down the load back to 600ops at time $t = 90$.

4.4.2 Results

BlowFish adaptation time and queue stability. As the load is increased from 600ops to 1800ops, the throughput supported by the shard at that storage ratio is insufficient to meet the increased load (Figures 9(a) and 9(d)). As a result, the request queue length for the shard increases (Figures 9(b) and 9(e)). At one point, BlowFish triggers *opportunistic layer creation* — the system immediately allocates additional storage for the two sampled arrays (increased storage ratio in Figures 9(c) and 9(f)); the sampled values are filled in gradually as queries are executed.

At this point, the results for uniform and skewed query distribution differ. For the uniform case, the already filled sampled values are reused infrequently. Thus, it takes BlowFish longer to adapt (≈ 5 minutes) before it starts draining the request queue (the peak in Figure 9(b)). BlowFish is able to drain the entire request queue within 15 minutes, making the system stable at that point.

For the skewed workload, the sampled values computed during query execution are reused frequently since queries repeat frequently. Thus, BlowFish is able to adapt much faster (≈ 2minutes) and drain the queues within 5 minutes. Note that this is akin to caching of results, explicitly implemented in many existing data stores [1, 4, 21] while BlowFish provides this functionality inherently.

BlowFish scheduling. To evaluate the effectiveness and stability of BlowFish scheduling, we turn our attention to a distributed setting. We focus our attention on three replicas of the same shard. We make the server storing one of these replicas storage constrained (replica #3); that is, irrespective of the load, the replica cannot trigger navigation along the storage-performance tradeoff curve. We then gradually increase the workload from 3KOps to 8KOps in steps of 1KOps per 30 minutes (Figure 10) and observe the behavior of request queues at the three replicas.

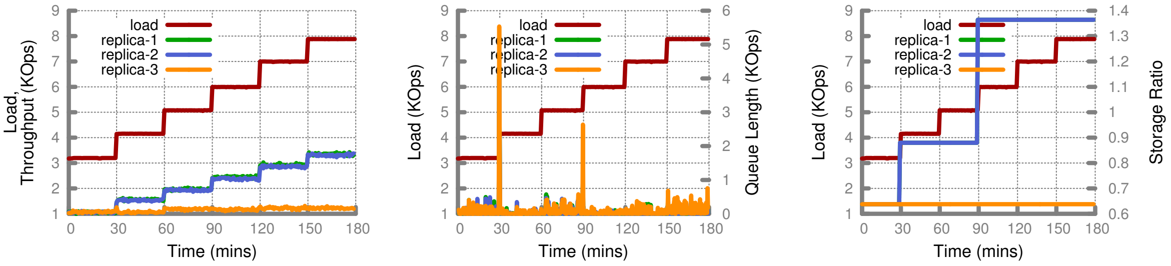


Figure 10: The effectiveness and stability of BlowFish's query scheduling mechanism in a replicated system (discussion in §4.4). Variation in throughput (left), request queue lengths (center) and storage-footprints (right) for the three replicas.

Initially, each of the three replicas observe a load of 1KOps since queue sizes are equal, and BlowFish scheduler equally balances the load. As the load is increased to 4KOps, the replicas are no longer able to match the load, causing the request queues at the replicas to build up (Figure 10(c)). Once the queue lengths cross the threshold, replica #1 and #2 trigger layer construction to match higher load (Figure 10(a)).

As the first two replicas opportunistically add layers, their throughput increases; however, the throughput for the third replicas remains consistent (Figure 10(b)). This causes the request queue to build up for the third replica at a rate higher than the other two replicas (Figure 10(c)). Interestingly, the BlowFish reduces quickly adapts, and stops issuing queries to replica#3, causing its request queue length to start dropping. We observe a similar trend when the load increases to 5KOps. BlowFish does observe queue length oscillations during adaptation, albeit of extremely small magnitude.

5 Related Work

BlowFish's goals are related to three key areas:

Storage-performance tradeoff. Existing data stores usually support two extreme operating points for each cached shard — compressed but low throughput, and uncompressed but high throughput. Several compression techniques (*e.g.*, gzip) can allow achieving different compression factors by changing parameters. However, these require decompression and re-compression of the entire data on the shard. As shown in the paper, a smooth and dynamic storage-performance tradeoff not only provides benefits for existing applications but can also enable a wide range of new applications.

Data repair. The tradeoff between known techniques for data repair — replication and erasure codes — is that of storage overhead and bandwidth. Studies have shown that the bandwidth requirement of traditional erasure codes is simply too high to use them in practice [29]. Several research proposals [20, 27, 29] reduce the bandwidth requirements of traditional erasure codes for batch processing jobs. However, these codes remain inefficient for data stores serving small objects. As shown in §4, BlowFish achieves storage close to erasure codes, while maintaining the bandwidth and repair time advantages of replication.

Selective Caching. As discussed in §1 and §4, selective caching can achieve good performance for workloads skewed towards a few popular objects. However, it only provides a coarse-grained support — increasing the throughput by 2× by increasing the storage overhead by 1×. BlowFish, instead, provides a much finer-grained control allowing applications to increase the storage fractionally, just enough to meet the performance goals.

6 Conclusion

BlowFish is a distributed data store that enables a smooth storage-performance tradeoff between two extremes — compressed but low throughput and uncompressed but high throughput. In addition, BlowFish allows applications to navigate along this tradeoff curve over fine-grained time scales. Using this flexibility, we explored several problems from real-world production clusters from a new "lens" and showed that the tradeoff exposed by BlowFish can offer significant benefits compared to state-of-the-art techniques for the respective problems.

Acknowledgments

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, Guavus, HP, Huawei, Intel, Microsoft, Pivotal, Samsung, Schlumberger, Splunk, State Farm, Virdata and VMware.

References

[1] Elasticsearch. http://www.elasticsearch.org.

[2] Longhair: Fast Cauchy Reed-Solomon Erasure Codes in C. https://github.com/catid/longhair.

[3] MemCached. http://www.memcached.org.

[4] MongoDB. http://www.mongodb.org.

[5] Redis. http://www.redis.io.

[6] TPC-H. http://www.tpc.org/tpch/.

[7] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling Queries on Compressed Data. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[8] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *ACM European Conference on Computer Systems (EuroSys)*, 2011.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64, 2012.

[10] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Technical Conference (ATC)*, 2013.

[12] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's Globally-distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[14] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[15] 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 *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[16] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[17] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012.

[18] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[19] V. Gupta, M. H. Balter, K. Sigman, and W. Whitt. Analysis of Join-the-Shortest-Queue Routing for Web Server Farms. 2007.

[20] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure Coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)*, 2012.

[21] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[22] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):1–10, 2009.

[23] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[24] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang,

and S. Kumar. f4: Facebook's Warm BLOB Storage System. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[25] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[26] A. Pavlo, C. Curino, and S. Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *ACM International Conference on Management of Data (SIGMOD)*, 2012.

[27] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2014.

[28] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[29] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. In *International Conference on Very Large Data Bases (VLDB)*, 2013.

[30] S. Sivasubramanian. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *ACM International Conference on Management of Data (SIGMOD)*, 2012.

[31] C. B. Walton, A. G. Dale, and R. M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *International Conference on Very Large Data Bases (VLDB)*, 1991.

A Layered Sampled Array Lookup, and Opportunistic layer creation

We outline how lookups are performed on the LSA (§3.1) in Algorithm 1. At a high level, given the LSA index, we obtain the layer ID and index into the corresponding layer using LSA's auxiliary structures (see Figure 4). We use the layer ID to locate the layer, and obtain the required value using the index into the layer.

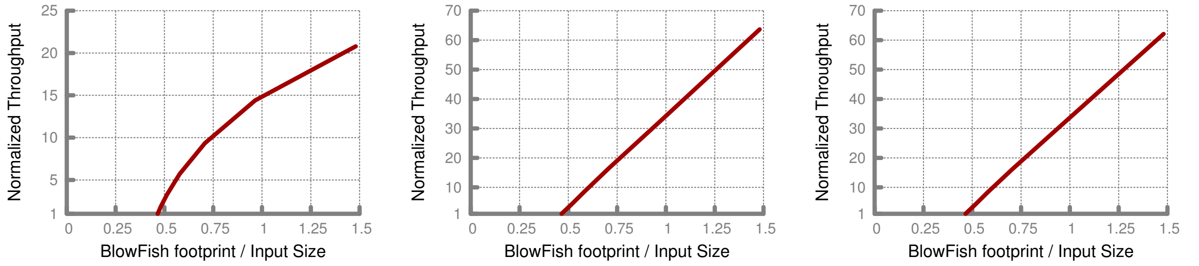
Algorithm 2 describes how BlowFish creates new layers *opportunistically* (§3.1); that is, rather than using dedicated resources to compute the required sampled values upon a new layer creation, BlowFish uses the computations performed during query execution to opportunistically populate the sampled values in the new layer.

Algorithm 1 LookupLSA

```
procedure GetLayerID (idx)    ▷ Get the layer ID given the index into the sampled array; $\alpha$ is the sampling rate.
    return LayerID[idx % $\alpha$]
end procedure
procedure GetLayerIdx(idx)    ▷ Get the index into LayerID given the index into the sampled array; $\alpha$ is the sampling rate.
    $count \leftarrow$ Count[LayerID(idx)]
    return $count \times$ (idx / $\alpha$) + LayerIdx[idx % $\alpha$]
end procedure
procedure LookupLSA (idx)    ▷ Performs lookup on the LSA.
    if IsSampled(idx) then
        $l_{id} \leftarrow$ GetLayerID(idx)    ▷ Get layer ID.
        $l_{idx} \leftarrow$ GetLayerIdx(idx)    ▷ Get index into layer.
        return SampledArray[$l_{id}$][$l_{idx}$]
    end if
end procedure
```

B Storage-throughput Tradeoff for different workloads

Figure 6 in §4 shows the storage-throughput tradeoff enabled by BlowFish for query workload comprising of 50% random access and 50% search queries. Figure 11 shows this tradeoff for other workloads. In particular, Figure 11(a) and Figure 11(b) show the storage-throughput tradeoff for workloads comprising of 100% random access and 100% search queries, respectively. Note that the tradeoff for mixed workload has characteristics similar to 100% `search` workload since, similar to other systems, execution time for search is significantly higher than random access. The throughput of the system is, thus, dominated by latency of search queries.



(a) 100% `random access` (b) 100% `search` (c) 50% `random access` + 50% `search`

Figure 11: Storage-throughput tradeoff curve (per thread) enabled by BlowFish for three workloads with varying fraction of `random access` and `search` queries. The y-axis is normalized by the throughput of smallest possible storage footprint in BlowFish (3874ops for random access only, 37ops for search only, and 71ops for the mixed workload).

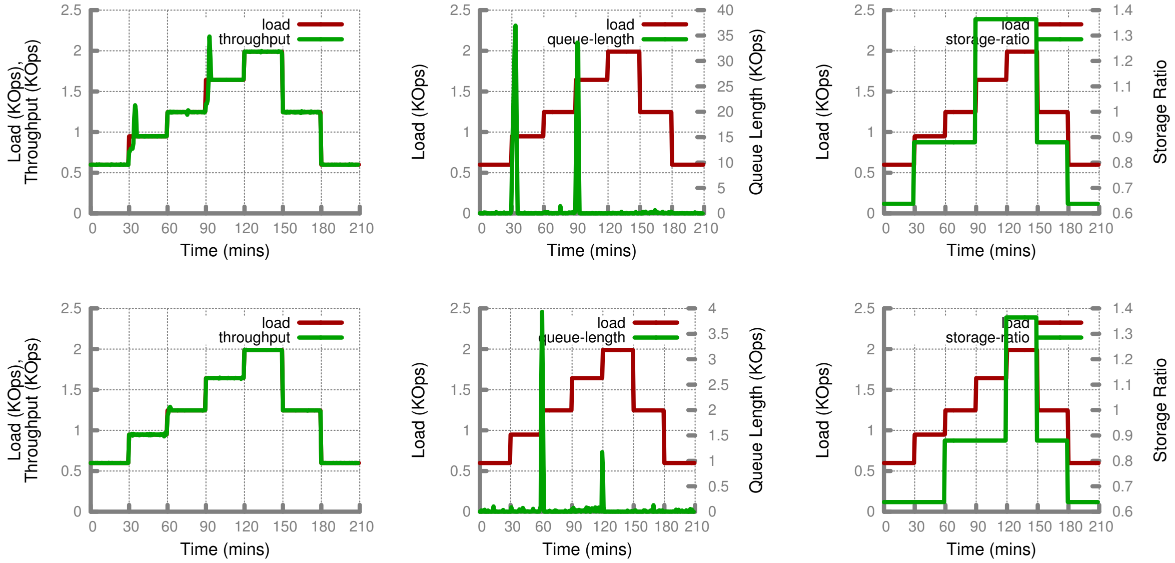


Figure 12: Opportunistic layer construction with gradual changes in load for uniform workload (top three) and skewed workload (bottom three). The figures show variation in throughput (left), request queue length (center) and storage footprint (right).

C Gradual Workload Variation

We present the results for how BlowFish adapts to time-varying workloads with a setup identical to §4.4, but for slightly different variations in the workload. In particular, instead of increasing the load on the shard from 600ops to 1800ops suddenly (as in the results of Figure 9), we increase the load from 600ops to 2000ops, with a gradual increase of 350ops at 30 minute intervals. This granularity of increase in load is similar to those reported in real-world production clusters [9], and constitutes a much easier case for BlowFish compared to the spiked increase in load considered in §4.4.

Uniform query distribution (Figure 12, top). As the load increases from 600ops to 950ops (Figure 12(a)), the load becomes higher than the throughput supported by the shard at that storage ratio (800ops). Consequently, the request queue length starts building up (Figure 12(b)), and BlowFish triggers a layer addition by allocating space for the new layers (Figure 12(c)). BlowFish opportunistically fills up values in the new layer, and the throughput for the shard increases gradually. This continues until the throughput matches the load on the shard; at this point, however, the throughput continues to increase even beyond the load to deplete the outstanding requests in the queue until the queue length reduces to zero and the system resumes normal operation. A similar trend can be seen when the load is increased to 1650ops.

Algorithm 2 `CreateLayerOpportunistic`

```
procedure CreateLayerOpportunistic($l_{id}$)    ▷ Marks layer $l_{id}$ for creation, and initializes bitmap marking layer's sampled values; $\alpha$ is the sampling rate.
    Mark layer $l_{id}$ for creation.
    LayerSize ← InputSize/2$\alpha$
    for $l_{idx}$ in (0, LayerSize − 1) do
        IsLayerValueSampled[$l_{id}$][$l_{idx}$] ← 0
    end for
end procedure
procedure OpportunisticPopulate(val, idx)    ▷ Exploit query execution to populate layers opportunistically; val is the unsampled values computed during query execution, and idx is its index into the unsampled array.
    $l_{id}$ ← GetLayerID(idx)    ▷ Get layer ID.
    if layer $l_{id}$ is marked for creation then
        $l_{idx}$ ← GetLayerIdx(idx)    ▷ Get index into layer.
        SampledArray[$l_{id}$][$l_{idx}$] ← $val$
        IsLayerValueSampled[$l_{id}$][$l_{idx}$] ← 1
    end if
end procedure
```

Skewed query distribution (Figure 12, bottom). The trends observed for the skewed workload are similar to those for the uniform worklod, with two key differences. First, we observe that BlowFish triggers layer creation at different points for this workload. In particular, the throughput for the skewed workload at the same storage footprint (0.8 in Figure 12(c) and 12(f)) is higher than that for the uniform workload. To see why, note that the performance of `search` operations varies significantly based on the queries; while the different queries contribute equally for the uniform workload, the throughput for the skewed workload is shaped by the queries that occur more frequently. This effect attributes for the different throughput characteristics for the two workloads at the same storage footprint.

Second, as noted before (§4.4), BlowFish adaptation benefits from the repetitive nature of queries in the skewed workload, since repeated queries can reuse the values populated during their previous execution. In comparison to uniform query distribution, this leads to faster adaptation to increase in load and quicker depletion of the increased request queue lengths.

Comparison with results for the spiked case. Note the difference in results for the case of spiked increase in load (Figure 9) and gradual increase in load (Figure 12). In the former case, the increase in load leads to significantly higher request queue lengths and hence, it takes much longer for the sytem to return to normal operations. In the latter, however, due to gradual increase in load, the system can drain the outstanding request queue significantly faster, can resume normal operations faster, and thus provides adaptation at much finer time granularity.

Universal Packet Scheduling

Radhika Mittal[†] Rachit Agarwal[†] Sylvia Ratnasamy[†] Scott Shenker[†‡]
[†]*UC Berkeley* [‡]*ICSI*

Abstract

In this paper we address a seemingly simple question: *Is there a universal packet scheduling algorithm?* More precisely, we analyze (both theoretically and empirically) whether there is a single packet scheduling algorithm that, at a network-wide level, can perfectly match the results of *any* given scheduling algorithm. We find that in general the answer is "no". However, we show theoretically that the classical Least Slack Time First (LSTF) scheduling algorithm comes closest to being universal and demonstrate empirically that LSTF can closely replay a wide range of scheduling algorithms. We then evaluate whether LSTF can be used *in practice* to meet various network-wide objectives by looking at popular performance metrics (such as average FCT, tail packet delays, and fairness); we find that LSTF performs comparable to the state-of-the-art for each of them. We also discuss how LSTF can be used in conjunction with active queue management schemes (such as CoDel and ECN) without changing the core of the network.

1 Introduction

There is a large and active research literature on novel packet scheduling algorithms, from simple schemes such as priority scheduling [38], to more complicated mechanisms for achieving fairness [20,34,39], to schemes that help reduce tail latency [19] or flow completion time [10], and this short list barely scratches the surface of past and current work. In this paper we do not add to this impressive collection of algorithms, but instead ask if there is a single *universal* packet scheduling algorithm that could obviate the need for new ones. In this context, we consider a packet scheduling algorithm to be both how packets are served inside the network (based on their arrival times and their packet headers) and how packet header fields are initialized and updated; this definition includes all the classical scheduling algorithms (FIFO, LIFO, priority, round-robin) as well as algorithms that incorporate dynamic packet state [19,44,45].

We can define a universal packet scheduling algorithm (hereafter UPS) in two ways, depending on our viewpoint on the problem. From a theoretical perspective, we call a packet scheduling algorithm *universal* if it can replay any *schedule* (the set of times at which packets arrive to and exit from the network) produced by any other scheduling algorithm. This is not of practical interest, since such schedules are not typically known in advance, but it offers a theoretically rigorous definition of universality that (as we shall see) helps illuminate its fundamental limits (i.e., which scheduling algorithms have the flexibility to serve as a UPS, and why).

From a more practical perspective, we say a packet scheduling algorithm is universal if it can achieve different desired performance objectives (such as fairness, reducing tail latencies and minimizing flow completion times). In particular, we require that the UPS should match the performance of the best known scheduling algorithm for a given performance objective. [1]

The notion of universality for packet scheduling might seem esoteric, but we think it helps clarify some basic questions. If there exists no UPS then we should *expect* to design new scheduling algorithms as performance objectives evolve. Moreover, this would make a strong argument for switches being equipped with programmable packet schedulers so that such algorithms could be more easily deployed (as argued in [42]; in fact, it was the eloquent argument in this paper that caused us to initially ask the question about universality).

However, if there is indeed a UPS, then it changes the lens through which we view the design and evaluation of scheduling algorithms: e.g., rather than asking whether a new scheduling algorithm meets a performance objective, we should ask whether it is easier/cheaper to implement/configure than the UPS (which could also meet that performance objective). Taken to the extreme, one might even argue that the existence of a (practical) UPS greatly diminishes the need for programmable *scheduling* hardware.[2] Thus, while the rest of the paper occasionally descends into scheduling minutiae, the question we are asking has important practical (and intriguing theoretical) implications.

This paper starts from the theoretical perspective, defining a formal model of packet scheduling and our

[1] For this definition of universality, we allow the header initialization to depend on the objective being optimized. That is, while the basic scheduling operations must remain constant, the header initialization can depend on whether you are seeking fairness or minimal flow completion time.

[2] Note that the case for programmable hardware as made in recent work on P4 and the RMT switch [14, 15] remains: these systems target programmability in header parsing and in how a packet's processing pipeline is defined (i.e., how forwarding 'actions' are applied to a packet). The P4 language does not currently offer primitives for scheduling and, perhaps more importantly, the RMT switch does not implement a programmable packet scheduler; we hope our results can inform the discussion on whether and how P4/RMT might be extended to support programmable scheduling.

notion of replayability in §2. We first prove that there is no UPS, but then show that Least Slack Time First (LSTF) [28] comes as close as any scheduling algorithm to achieving universality. We also demonstrate empirically (via simulation) that LSTF can closely approximate the schedules of many scheduling algorithms. Thus, while not a perfect UPS in terms of replayability, LSTF comes very close to functioning as one.

We then take a more practical perspective in §3, showing (via simulation) that LSTF is comparable to the state of the art in achieving various objectives relevant to an application's performance. We investigate in detail LSTF's ability to minimize average flow completion times, minimize tail latencies, and achieve per-flow fairness. We also consider how LSTF can be used in multitenant situations to achieve multiple objectives simultaneously, while highlighting some of its key limitations.

In §4, we look at how network feedback for active queue management (AQM) can be incorporated using LSTF. Rather than augmenting the basic LSTF logic (which is restricted to packet scheduling) with a queue management algorithm, we show that LSTF can, instead, be used to implement AQM at the edge of the network. This novel approach to AQM is a contribution in itself, as it allows the algorithm to be upgraded without changing internal routers.

We then discuss the feasibility of implementing LSTF (§5) and provide an overview of related work (§6) before concluding with a discussion of open questions in §7.

2 Theory: Replaying Schedules

This section delves into the theoretical viewpoint of a UPS, in terms of its ability to *replay* a given schedule.

2.1 Definitions and Overview

Network Model: We consider a network of store-and-forward output-queued routers connected by links. The input load to the network is a fixed set of packets $\{p \in P\}$, their arrival times $i(p)$ (*i.e.,* when they reach the ingress router), and the path $path(p)$ each packet takes from its ingress to its egress router. We assume no packet drops, so all packets eventually exit. Every router executes a non-preemptive scheduling algorithm which need not be work-conserving or deterministic and may even involve oracles that know about future packet arrivals. Different routers in the network may use different scheduling logic. For each incoming load $\{(p,i(p),path(p))\}$, a collection of scheduling algorithms $\{A_\alpha\}$ (router α implements algorithm A_α) will produce a set of packet output times $\{o(p)\}$ (the time a packet p exits the network). We call the set $\{(path(p),i(p),o(p))\}$ a *schedule*.

Replaying a Schedule: Applying a different collection of scheduling algorithms $\{A'_\alpha\}$ to the same set of packets $\{(p,i(p),path(p))\}$ (with the packets taking the same path in the replay as in the original schedule), produces a new set of output times $\{o'(p)\}$. We say that $\{A'_\alpha\}$ *replays* $\{A_\alpha\}$ on this input if and only if $\forall p \in P$, $o'(p) \leq o(p)$.[3]

Universal Packet Scheduling Algorithm: We say a schedule $\{(path(p),i(p),o(p))\}$ is *viable* if there is at least one collection of scheduling algorithms that produces that schedule. We say that a scheduling algorithm is *universal* if it can replay *all* viable schedules. While we allowed significant generality in defining the scheduling algorithms that a UPS seeks to replay (demanding only that they be non-preemptive), we insist that the UPS itself obey several practical constraints (although we allow it to be preemptive for theoretical analysis, but then quantitatively analyze the non-preemptive version in §2.3).[4] The three practical constraints we impose on a UPS are:

(1) Uniformity and Determinism: A UPS must use the same deterministic scheduling logic at every router.

(2) Limited state used in scheduling decisions: We restrict a UPS to using only (i) packet headers, and (ii) static information about the network topology, link bandwidths, and propagation delays. It cannot rely on oracles or other external information. However, it can modify the header of a packet before forwarding it (resulting in *dynamic packet state* [45]).

(3) Limited state used in header initialization: We assume that the header for a packet p is initialized at its ingress node. The additional information available to the ingress for this initialization is limited to: (i) $o(p)$ from the original schedule[5] and (ii) $path(p)$. Later, we extend the kinds of information the header initialization process can use, and find that this is a key determinant in whether one can find a UPS.

We make three observations about the above model. First, our model assumes greater capability at the edge than in the core, in keeping with common assumptions that the network edge is capable of greater processing complexity, exploited by many architectural proposals [16,36,44]. Second, when initializing a packet p's header, a UPS can only use the input time, output time and the path information for p itself, and must be *oblivious* [24] to the corresponding attributes for *other* packets in the network. Finally, the key source of impracticality in our model is the assumption that the output times $o(p)$ are known at the ingress. However,

[3] We allow the inequality because, if $o'(p) < o(p)$, one can delay the packet upon arrival at the egress node to ensure $o'(p) = o(p)$.

[4] The issue of preemption is somewhat complicated. Allowing the original scheduling algorithms to be preemptive allows packets to be fragmented, which then makes replay extremely difficult even in simple networks (with store-and-forward routers). However, disallowing preemption in the candidate UPS overly limits the flexibility and would again make replay impossible even in simple networks. Thus, we take the seemingly hypocritical but only theoretically tractable approach and disallow preemption in the original scheduling algorithms but allow preemption in the candidate UPS. In practice, when we care only about approximately replaying schedules, the distinction is of less importance, and we simulate LSTF in the non-preemptive form.

[5] Note that this ingress router can directly observe $i(p)$ as the time the packet arrives.

a different interpretation of $o(p)$ suggests a more practical application of replayability (and thus our results): *if we assign $o(p)$ as the "desired" output time for each packet in the network, then the existence of a UPS tells us that if these goals are viable then the UPS will be able to meet them.*

2.2 Theoretical Results

For brevity, in this section we only summarize our key theoretical results. The detailed proofs are in Appendix A.

Existence of a UPS under omniscient initialization: Suppose we give the header-initialization process extensive information in the form of times $o(p,\alpha)$ which represent when p was scheduled by router α in the original schedule. We can then insert an n-dimensional vector in the header of every packet p, where the i^{th} element contains $o(p,\alpha_i)$ with α_i being the i^{th} hop in $path(p)$. Every time a packet arrives at a router, the router can pop the value at the head of this vector and use that as its priority (earlier values of output times get higher priority). This can perfectly replay any viable schedule (proof in Appendix A.2), which is not surprising, as having such detailed knowledge of the internal scheduling of the network is tantamount to knowing all the scheduling decisions made by the original algorithm. For reasons discussed previously, our definition limited the information available to the output time from the network as a whole, and not from each individual router; we call this *black-box* initialization.

Nonexistence of a UPS under black-box initialization: We can prove by counter-example (described in Appendix A.3) that *there is no UPS* under the conditions stated in §2.1. We provide some intuition for the counter-example later in this section. Given this impossibility result, we now ask *how close can we get to a UPS?*

Natural candidates for a near-UPS: Simple priority scheduling [6] can reproduce all viable schedules on a single router, so it would seem to be a natural candidate for a near-UPS. However, for multihop networks it may be important to make the scheduling of a packet dependent on what has happened to it earlier in its path. For this, we consider Least Slack Time First (LSTF) [28].

In LSTF, each packet p carries its slack value in the packet header, which is initialized to $slack(p) = (o(p) - i(p) - t_{min}(p,src(p),dest(p)))$ at the ingress; where $src(p)$ is the ingress of p, $dest(p)$ is the egress of p and $t_{min}(p,\alpha,\beta)$ is the time p takes to go from router α to router β in an uncongested network. Therefore, the slack of a packet indicates the maximum queueing time (excluding the transmission time at any router) that the packet could tolerate without violating the replay condition. Each router, then, schedules the packet which has the least remaining slack at the time when its last bit is transmitted. Before forwarding the packet, the router overwrites the slack value in the packet's header with its remaining slack (*i.e.,* the previous slack time minus the duration for which it waited in the queue before being transmitted).

An alternate way to implement this algorithm is having a static packet header as in Earliest Deadline First (EDF) and using additional state in the routers (reflecting the value of t_{min}) to compute the priority for a packet at each router, but here we chose to use an approach with dynamic packet state. We provide more details about EDF and prove its equivalence to LSTF in Appendix A.5.

Key Results: Our analysis shows that the difficulty of replay is determined by the number of *congestion points*, where a *congestion point* is defined as a node where a packet is forced to "wait" during a given schedule. [7] Our theorems show the following key results:
1. Priority scheduling can replay all viable schedules with no more than one congestion point per packet, and there are viable schedules with no more than two congestion points per packet that it cannot replay. (Proof in Appendix A.6.)
2. LSTF can replay all viable schedules with no more than two congestion points per packet, and there are viable schedules with no more than three congestion points per packet that it cannot replay. (Proof in Appendix A.7.)
3. There is no scheduling algorithm (obeying the aforementioned constraints on UPSs) that can replay *all* viable schedules with no more than three congestion points per packet, and the same holds for larger numbers of congestion points. (Proof in Appendix A.3.)

Main Takeaway: *LSTF is closer to being a UPS than simple priority scheduling, and no other candidate UPS can do better in terms of handling more congestion points.*

Intuition: It is clear why LSTF is superior to priority scheduling: by carrying information about previous delays in the packet header (in the form of the *remaining* slack value), LSTF can "make up for lost time" at later congestion points, whereas for priority scheduling packets with low priority might repeatedly get delayed (and thus miss their target output times).

We now provide some intuition for why LSTF works for two congestion points and not for three, by presenting an outline of the proof detailed in Appendix A.7. We define the *local deadline* of a packet p at a router α as the time when p is scheduled by α in the original schedule. The *global deadline* of p at α is defined as the time by when p must leave α

[6] By simple priority scheduling, we mean that the ingress assigns priority values to the packets and the routers simply schedule packets based on these static priority values.

[7] For our theoretical results, we adopt a pessimistic definition of a congestion point, where a router that falls in the path of more than one flow is a congestion point (along with routers having output link capacity less than input link capacity or non work-conserving original schedules that make a packet wait explicitly). Since this definition is independent of per-packet dynamics, the set of congestion points remains the same in the original schedule and in the replay. This pessimistic definition is not required in practice, where the difficulty of replay would depend on the number of routers in a packet's path which see significant queuing.

in order to meet its target output time, assuming that it sees no queuing delay after α. Hence, *global deadline* is the time when p's slack at α becomes zero. We can prove that *as long as all packets arrive at a router at or before their local deadlines during the LSTF replay, no packet can miss its global deadline at α (i.e. no packet can have a negative slack at α)*. The proof for this follows from the fact that if all packets arrive at or before their *local deadline* at α, there exists a *feasible* schedule where no packet misses its *global deadline* at α (this *feasible* schedule is the same as the original schedule at α). We can now apply the standard LSTF (or EDF) optimality proof technique for a single processor [30], to show that this feasible schedule can be iteratively *transformed* to a feasible LSTF schedule at router α.

When there are only two congestion points per packet, it is guaranteed that every packet arrives at or before its local deadline at each congestion point during the LSTF replay. A packet can never arrive after its *local deadline* at its first congestion point, because it sees no queuing before that. Moreover, the *local deadline* is the same as the *global deadline* at the last congestion point. Therefore, if a packet arrives after its *local deadline* at its second (and last) congestion point, it means that it must have already missed its *global deadline* earlier, which, again, is not possible.

However, when there are three congestion points per packet, there is no guarantee that every packet arrives at or before its local deadline at each congestion point during the LSTF replay (due to the presence of a "middle" congestion point). One can, therefore, create counterexamples where unless LSTF (or, in fact, any other scheduling algorithm) makes precisely the right choice at the first congestion point of a packet p, at least one packet will miss its target output time, due to p arriving after its *local deadline* at its middle congestion point. We present such a counterexample in Appendix A.3, where we illustrate two ways of scheduling the same set of packets (having the same input times and paths) on a given topology with three congestion points per packet, resulting in two cases. The output times for two of the packets (named a and x), which compete with each other at the first congestion point (α_0), remains the same in both cases. However, one case requires scheduling a before x at α_0 and the second case requires scheduling x before a at α_0, else a packet will end up missing its target output time at the second (or middle) congestion points of a and x respectively. Since the information available for header initialization for the two packets is the same in both cases, no deterministic scheduling algorithm with blackbox header initialization can make the *correct* choice at the first congestion point *in both cases*.

2.3 Empirical Results

The previous section clarified the theoretical limits on a *perfect* replay. Here we investigate, via ns-2 simulations [6], how well (a non-preemptable version of) LSTF can *approximately* replay schedules in realistic networks.

Experiment Setup: *Default scenario.* We use a simplified Internet-2 topology [3], identical to the one used in [31] (consisting of 10 core routers connected by 16 links). We connect each core router to 10 edge routers using 1Gbps links and each edge router is attached to an end host via a 10Gbps link. The number of hops per packet is in the range of 4 to 7, excluding the end hosts. We refer to this topology as I2 1Gbps-10Gbps. Each end host generates UDP flows using a Poisson inter-arrival model, with the destination picked randomly for each flow. Our default scenario runs at 70% utilization. The flow sizes are picked from a heavy-tailed distribution [11, 12]. Since our focus is on packet scheduling, not dropping policies, we use large buffer sizes that ensure no packet drops. Note that we use higher than usual access bandwidths for our default scenario to increase the stress on the schedulers in the core routers, where the number of congestion points seen by most packets is two, three or four for 22%, 44% and 24% packets respectively. [8] We also present results for smaller (and more realistic) access bandwidths, where most packets see smaller number of congestion points (one, two or three for 18%, 46% and 26% packets respectively), resulting in better replay performance.

Varying parameters. We tested a wide range of experimental scenarios by varying different parameters from their default values. We present results for a small subset of these scenarios here: (1) the default scenario with network utilization varied from 10-90% (2) the default scenario but with 1Gbps link between the endhosts and the edge routers (I2 1Gbps-1Gbps), with 10Gbps links between the edge routers and the core (I2 10Gbps-10Gbps) and with all link capacities in the I2 1Gbps-1Gbps topology reduced by a factor of 10 (I2 / 10) and (3) the default scenario applied to two different topologies, a bigger Rocketfuel topology [43] (with 83 core routers connected by 131 links) and a full bisection bandwidth datacenter (fat-tree) topology from [10] (with 10Gbps links). Note that our other results were generally consistent with those presented here.

Scheduling algorithms. Our default case, which we expected to be hard to replay, uses completely arbitrary schedules produced by a *random* scheduler (which picks the packet to be scheduled randomly from the set of queued up packets). We also present results for more traditional packet scheduling algorithms: FIFO, LIFO, fair queuing [20], and SJF (shortest job first using priorities). We also looked at two scenarios with a mixture of scheduling algorithms: one where half of the routers run FIFO+ [19] and the other half run fair queuing, and one where fair queueing is used to isolate two classes of traffic, with one class being scheduled with SJF and the

[8] To compute this, we record the number of non-empty queues (excluding the endhost queues) encountered by each packet.

other class being scheduled with FIFO.

Topology	Avg. Link Utilization	Scheduling Algorithm	Fraction of packets overdue	
			Total	$>T$
I2 1Gbps-10Gbps	70%	Random	0.0021	0.0002
I2 1Gbps-10Gbps	10%	Random	0.0007	0.0
	30%		0.0281	0.0017
	50%		0.0221	0.0002
	90%		0.0008	4×10^{-6}
I2 1Gbps-1Gbps	70%	Random	0.0204	8×10^{-6}
I2 10Gbps-10Gbps			0.0631	0.0448
I2 / 10			0.0127	0.00001
Rocketfuel	70%	Random	0.0246	0.0063
Datacenter			0.0164	0.0154
I2 1Gbps-10Gbps	70%	FIFO	0.0143	0.0006
		FQ	0.0271	0.0002
		SJF	0.1833	0.0019
		LIFO	0.1477	0.0067
		FQ/FIFO+	0.0152	0.0004
		FQ: SJF/FIFO	0.0297	0.0003

Table 1: LSTF replay performance across various scenarios. T represents the transmission time at the bottleneck link.

Evaluation Metrics: We consider two metrics. First, we measure the fraction of packets that are overdue (i.e., which do not meet the original schedule's target). Second, to capture the *extent* to which packets fail to meet their targets, we measure the fraction of packets that are overdue by more than a threshold value T, where T is one transmission time on the bottleneck link ($\approx 12\mu s$ for 1Gbps). We pick this value of T both because it is sufficiently small that we can assume being overdue by this small amount is of negligible practical importance, and also because this is the order of violation we should expect given that our implementation of LSTF is non-preemptive. While we may have many small violations of replay (because of non-preemption), one would hope that most such violations are less than T.

Results: Table 1 shows the simulation results for LSTF replay for various scenarios, which we now discuss.

(1) Replayability. Consider the column showing the fraction of packets overdue. In all but three cases (we examine these shortly) over 97% of packets meet their target output times. In addition, the fraction of packets that did not arrive within T of their target output times is much smaller; even in the worst case of SJF scheduling (where 18.33% of packets failed to arrive by their target output times), only 0.19% of packets are overdue by more than T. Most scenarios perform substantially better: e.g., in our default scenario with Random scheduling, only 0.21% of packets miss their targets and only 0.02% are overdue by more than T. Hence, we conclude that even without preemption LSTF achieves good (but not perfect) replayability under a wide range of scenarios.

(2) Effect of varying network utilization. The second row in Table 1 shows the effect of varying network utilization. We see that at low utilization (10%), LSTF achieves exceptionally good replayability with a total of only 0.07% of packets overdue. Replayability deteriorates as utilization is increased to 30% but then (somewhat surprisingly) improves again as utilization increases. This improvement occurs because with increasing utilization, the amount of queuing (and thus the average slack across packets) in the original schedule also increases. This provides more room for slack re-adjustments when packets wait longer at queues seen early in their paths during the replay. We observed this trend in all our experiments though the exact location of the "low point" varied across settings.

(3) Effect of varying link bandwidths. The third row shows the effect of changing the relative values of access/edge *vs.* core links. We see that while decreasing access link bandwidth (I2 1Gbps-1Gbps) resulted in a much smaller fraction of packets being overdue by more than T (0.0008%), increasing the edge-to-core link bandwidth (I2 10Gbps-10Gbps) resulted in a significantly higher fraction (4.48%). For I2 1Gbps-1Gbps, packets are paced by the endhost link, resulting in few congestion points thus improving LSTF's replayability. In contrast, with I2 10Gbps-10Gbps, both the access and edge links have a higher bandwidth than most core links; hence packets (that are no longer paced at the endhosts or the edges) arrive at the core routers very close to one another and hence the effect of one packet being overdue *cascades* over to the following packets. Decreasing the absolute bandwidths in I2 / 10, while keeping the ratio between access and edge links the same as that in I2 1Gbps-1Gbps, did not produce significantly different results compared to I2 1Gbps-1Gbps, indicating that the relative link capacities have a greater impact on the replay performance than the absolute link capacities.

(4) Effect of varying topology. The fourth row in Table 1 shows our results using different topologies. LSTF performs well in both cases: only 2.46% (Rocketfuel) and 1.64% (datacenter) of packets fail replay. These numbers are still somewhat higher than our default case. The reason for this is similar to that for the I2 10Gbps-10Gbps topology – all links in the datacenter fat-tree topology are set to 10Gbps, while in our simulations, we set half of the core links in the Rocketfuel topology to have bandwidths smaller than the access links.

(5) Varying Scheduling Algorithms. Row five in Table 1 shows LSTF's ability to replay different scheduling algorithms. We see that LSTF performs well for FIFO, FQ, and the combination cases (a mixture of FQ/FIFO+ and having FQ share between FIFO and SJF); e.g., with FIFO, fewer than 0.06% of packets are overdue by more than T. However, there are two problematic cases: SJF and LIFO fare worse with 18.33% and 14.77% of packets failing replay (although only 0.19% and 0.67% of packets are overdue by more than T respectively). The reason stems from a combination of two factors: (1) for these algorithms a larger fraction of packets have a very small slack value (as one might expect from the scheduling logic which

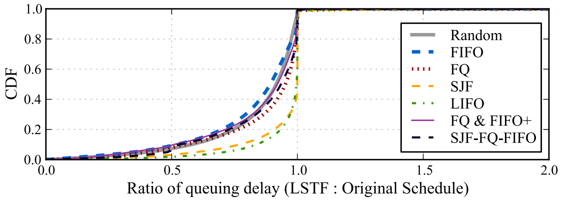


Figure 1: Ratio of queuing delay with varying packet scheduling algorithms, on I2 1Gbps-10Gbps topology at 70% utilization.

produces a larger skew in the slack distribution), and (2) for these packets with small slack values, LSTF *without preemption* is often unable to "compensate" for misspent slack that occurred earlier in the path. To verify this intuition, we extended our simulator to support preemption and repeated our experiments: with preemption, the fraction of packets that failed replay dropped to 0.24% (from 18.33%) for SJF and to 0.25% (from 14.77%) for LIFO.

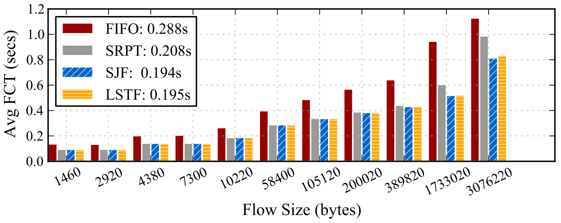
(6) End-to-end (Queuing) Delay. Our results so far evaluate LSTF in terms of measures that we introduced to test universality. We now evaluate LSTF using the more traditional metric of packet delay, focusing on the queueing delay a packet experiences. Figure 1 shows the CDF of the ratios of the queuing delay that a packet sees with LSTF to the queuing delay that it sees in the original schedule, for varying packet scheduling algorithms. We were surprised to see that most of the packets actually have a smaller queuing delay in the LSTF replay than in the original schedule. This is because LSTF eliminates "wasted waiting", in that it never makes packet A wait behind packet B if packet B is going to have significantly more waiting later in its path.

(7) Comparison with Priorities. To provide a point of comparison, we also did a replay using simple priorities for our default scenario, where the priority for a packet p is set to $o(p)$ (which seemed most intuitive to us). As expected, the resulting replay performance is much worse than LSTF: 21% packets are overdue in total, with 20.69% being overdue by more than T. For the same scenario, LSTF has only 0.21% packets overdue in total, with merely 0.02% packets overdue by more than T.

Summary: We observe that, in almost all cases, less than 1% of the packets are overdue with LSTF by more than T. The replay performance initially degrades and then starts improving as the network utilization increases. The distribution of link speeds has a bigger influence on the replay results than the scale of the topology. Replay performance is better for scheduling algorithms that produce a smaller skew in the slack distribution. LSTF replay performance is significantly better than simple priorities replay performance, with the most intuitive priority assignment.

3 Practical: Achieving Various Objectives

While replayability demonstrates the theoretical flexibility of LSTF, it does not provide evidence that it would be practically useful. In this section we look at how LSTF can be used *in practice* to meet the following performance objectives: minimizing average flow completion times, minimizing tail latencies, and achieving per-flow fairness.



Expt. Setup	Avg FCT (s)			
	FIFO	SRPT	SJF	LSTF
I2 1Gbps-10Gbps at 30% util.	0.189	0.183	0.182	0.182
I2 1Gbps-10Gbps at 50% util.	0.212	0.189	0.185	0.185
I2 1Gbps-10Gbps at 70% util.	0.288	0.208	0.194	0.195
I2 1Gbps-1Gbps at 70% util.	0.252	0.209	0.202	0.202
I2 / 10 at 70% util.	0.899	0.658	0.620	0.621
Rocketfuel at 70% util.	0.305	0.240	0.228	0.228
Datacenter at 70% util.	0.058	0.018	0.016	0.015

Figure 2: The graph shows the average FCT bucketed by flow size obtained with FIFO, SRPT and SJF (using priorities and LSTF) for I2 1Gbps-10Gbps at 70% utilization. The legend indicates the average FCT across all flows. The table indicates the average FCTs for varying settings.

Since the knowledge of a previous schedule is unavailable in practice, instead of using a given set of output times (as done in §2.3), we now use heuristics to assign the slacks in an effort to achieve these objectives. Our goal here is not to outperform the state-of-the-art for each objective in all scenarios, but instead we aim to be competitive with the state-of-the-art in most common cases.

In presenting our results for each objective, we first describe the slack initialization heuristic we use and then present some ns-2 [6] simulation results on (i) how LSTF performs relative to the state-of-the-art scheduling algorithm and (ii) how they both compare to FIFO scheduling (as a baseline to indicate the overall impact of specialized scheduling for this objective). As our default case, we use the I2 1Gbps-10Gbps topology using the same workload as in the previous section (running at 70% average utilization). We also present aggregate results at different utilization levels and for variations in the default topology (I2 1Gbps-1Gbps and I2 / 10), for the bigger Rocketfuel topology, and for the datacenter topology (for selected objectives). The switches use non-preemptive scheduling (including for LSTF) and have finite buffers (packets with the highest slack are dropped when the buffer is full). Unless otherwise specified, our experiments use TCP flows with router buffer sizes of 5MB for the WAN simulations (equal to the average bandwidth-delay product for our default topology) and 500KB for the datacenter simulations.

3.1 Average Flow Completion Time

While there have been several proposals on how to minimize flow completion time (FCT) via the transport protocol [21, 31], here we focus on *scheduling*'s impact on FCT, while using standard TCP New Reno at the endhosts. In [10] it is shown that (i) Shortest Remaining Processing

Time (SRPT) is close to optimal for minimizing the mean FCT and (ii) Shortest Job First (SJF) produces results similar to SRPT for realistic heavy-tailed distribution. Thus, these are the two algorithms we use as benchmarks.

Slack Initialization: We make LSTF emulate SJF by initializing the slack for a packet p as $slack(p) = fs(p) * D$, where $fs(p)$ is the size of the flow to which p belongs (in terms of the number of MSS-sized packets in the flow) and D is a value much larger than the queuing delay seen by any packet in the network. We use a value of $D = 1$ sec for our simulations.

Evaluation: Figure 2 compares LSTF with three other scheduling algorithms – FIFO, SJF and SRPT with *starvation prevention* as in [10]. Both SJF and SRPT have significantly lower mean FCT than FIFO. The LSTF based execution of SJF produces nearly the same results as the strict priorities based execution.

We also look at how in-network scheduling can be used along with changes in the endhost TCP stack to achieve the same objective in Appendix B.

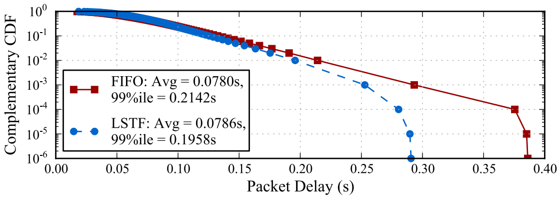
3.2 Tail Packet Delays

Clark et. al. [19] proposed the FIFO+ algorithm, where packets are prioritized at a router based on the amount of queuing delay they have seen at their previous hops, for minimizing the tail packet delays in multi-hop networks. FIFO+ is *identical* to LSTF scheduling where all packets are initialized with the same slack value.

Slack Initialization: All incoming packets are initialized with the same slack value (we use an initial slack value of 1 second in our simulations). With the slack update taking place at every router, the packets that have waited longer in the network queues are naturally given preference over those that have waited for a smaller duration.

Evaluation: We compare LSTF (which, with the above slack initialization, is identical to FIFO+) with FIFO, the primary metric being the 99%ile end-to-end one way delay seen by the packets. Figure 3 shows our results. To better understand the impact of the two scheduling policies on the packet delays, our evaluation uses an open-loop setting with UDP flows. With LSTF, packets that have traversed through more number of hops, and have therefore spent more slack in the network, get preference over shorter-RTT packets that have traversed through fewer hops. While this might produce a slight increase in the average packet delay, it reduces the tail. This is in-line with the observations made in [19].

3.3 Fairness



Expt. Setup	Avg Delay (s)		99%ile Delay (s)	
	FIFO	LSTF	FIFO	LSTF
I2 1Gbps-10Gbps at 30% util.	0.0411	0.0411	0.0911	0.0868
I2 1Gbps-10Gbps at 50% util.	0.0516	0.0517	0.1288	0.1195
I2 1Gbps-10Gbps at 70% util.	0.0780	0.0786	0.2142	0.1958
I2 1Gbps-1Gbps at 70% util.	0.0771	0.0771	0.2163	0.216
I2 / 10 at 70% util.	0.5762	0.5765	1.9393	1.9367
Rocketfuel at 70% util.	0.1891	0.1883	3.8139	3.7199
Datacenter at 70% util.	0.0250	0.0240	0.1352	0.1100

Figure 3: Tail packet delays for LSTF compared to FIFO. The graph shows the complementary CDF of packet delays for the I2 1Gbps-10Gbps topology at 70% utilization with the average and 99%ile packet delay values indicated in the legend. The table shows the corresponding results for varying settings.

Fairness is a common scheduling goal, which involves two different aspects: *asymptotic* bandwidth allocation (eventual convergence to the fair-share rate) and *instantaneous* bandwidth allocation (enforcing this fairness on small time-scales, so every flow experiences the equivalent of a per-flow pipe). The former can be measured by looking at long-term throughput measures, while the latter is best measured in terms of the flow completion times of relatively short flows (which measures bandwidth allocation on short time scales). We now show how LSTF can be used to achieve both of these goals, but more effectively the former than the latter. Our slack assignment heuristic can also be easily extended to achieve weighted fair queuing, but we do not present those results here.

Slack Initialization: The slack assignment for fairness works on the assumption that we have some ballpark notion of the fair-share rate for each flow and that it does not fluctuate wildly with time. Our approach to assigning slacks is inspired from [46]. We assign $slack = 0$ to the first packet of the flow and the slack of any subsequent packet p_i is then initialized as:

$$slack(p_i) = max\Big(0, slack(p_{i-1}) + \frac{size(p_i)}{r_{est}} - \big(i(p_i) - i(p_{i-1})\big)\Big)$$

where $i(p)$ is the arrival time of a packet p at the ingress, $size(p)$ is its size in bits, and r_{est} is an estimate of the fair-share rate r^* in bps. We show that the above heuristic leads to asymptotic fairness, for *any* value of r_{est} that is less than r^*, as long as all flows use the same value. The same heuristic can also be used to provide instantaneous fairness, when we have a complex mix of short-lived flows, where the r_{est} value that performs the best depends on the link bandwidths and their utilization levels. A reasonable value of r_{est} can be estimated using knowledge about the network topology and traffic matrices, though we leave a detailed exploration of this to future work.

Evaluation: Asymptotic Fairness. We evaluate the asymptotic fairness property by running our simulation on the Internet2 topology with 10Gbps edges, such that all the congestion happens at the core. However, we reduce the propagation delay to 10μs for each link, to make

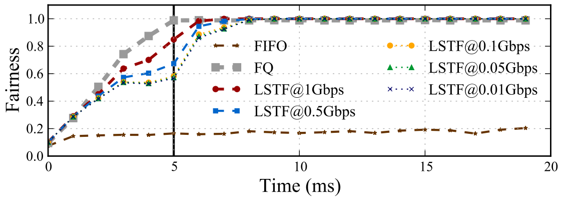


Figure 4: Fairness for long-lived flows on Internet2 topology. The legend indicates the value of r_{est} used for LSTF slack initialization.

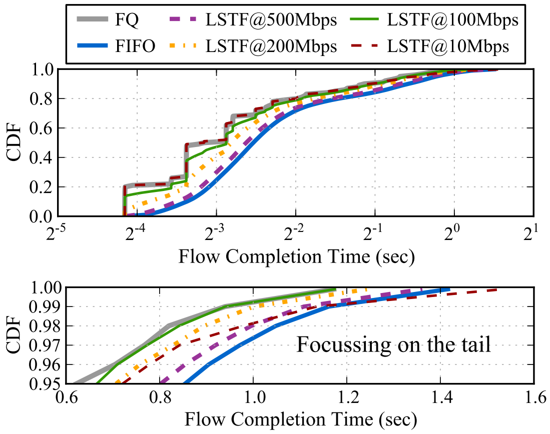


Figure 5: CDF of FCTs for the I2 1Gbps-10Gbps topology at 70% utilization.

Expt. Setup	Avg FCT across bytes (s)			Best r_{est} (Mbps)	Reasonable r_{est} Range (Mbps)
	FIFO	FQ	LSTF		
I2 1Gbps-10Gbps at 30% util.	0.563	0.537	0.538	300	10-900
I2 1Gbps-10Gbps at 50% util.	0.626	0.549	0.555	200	10-800
I2 1Gbps-10Gbps at 70% util.	0.811	0.622	0.632	100	50-200
I2 1Gbps-1Gbps at 70% util.	0.766	0.630	0.652	100	50-400
I2 / 10 at 70% util.	4.838	2.295	2.759	10	10-20
Rocketfuel at 70% util.	0.964	0.796	0.824	100	50-300

Table 2: FCT averaged across bytes for FIFO, FQ and LSTF (with best r_{est} value) across varying settings. The last column indicates the range of r_{est} values that produce results within 10% of the best r_{est} result.

the experiment more scalable, while the buffer size is kept large (50MB) so that fairness is dominated by the scheduling policy and not by how TCP reacts to packet drops. We start 90 long-lived TCP flows with a random jitter in the start times ranging from 0-5ms. The topology is such that the fair share rate of each flow on each link in the core network (which is shared by up to 13 flows) is around 1Gbps. We use different values for $r_{est} \leq 1$Gbps for computing the initial slacks and compare our results with fair queuing (FQ). Figure 4 shows the fairness computed using Jain's Fairness Index [27], from the throughput each flow receives per millisecond. Since we use the throughput received by each of the 90 flows to compute the fairness index, it reaches 1 with FQ only at 5ms, after all the flows have started. We see that LSTF is able to converge to perfect fairness, even when r_{est} is 100X smaller than r^*. It converges slightly sooner when r_{est} is closer to r^*, though the subsequent differences in the time to convergence decrease with decreasing values of r_{est}.

The detailed explanation of how this works along with more evaluation (on multiple bottlenecks and weighted fairness) has been provided in Appendix C.

Evaluation: Instantaneous Fairness. As one might expect, the choice of r_{est} has a bigger impact on instantaneous fairness than on asymptotic fairness. A very high r_{est} value would not provide sufficient isolation across flows. On the other hand, a very small r_{est} value can starve the long flows. This is because the assigned slack values for the later packets of long flows with high sequence numbers would be much higher than the actual slack they experience. As a result, they will end up waiting longer in the queues, while the initial packets of newer flows with smaller slack values would end up getting a higher precedence.

To verify this intuition, we evaluated our LSTF slack assignment scheme by running our standard workload with a mix of TCP flows ranging from sizes 1.5KB - 3MB on our default I2 1Gbps-10Gbps topology at 70% utilization, with 50MB buffer size. Note that the traffic pattern is now bursty and the instantaneous utilization of a link is often lower or higher than the assigned average utilization level. The CDF of the FCTs thus obtained is shown in Figure 5. As expected, the distribution of FCTs looks very different between FQ and FIFO. FQ isolates the flows from each-other, significantly reducing the FCT seen by short to medium size flows, compared to FIFO. The long flows are also helped a little by FQ, again due to the isolation provided from one-another.

LSTF performance varies somewhere in between FIFO and FQ, as we vary r_{est} values between 500Mbps to 10Mbps. A high value of r_{est} = 500Mbps does not provide sufficient isolation and the performance is close to FIFO. As we reduce the value of r_{est}, the "isolation-effect" increases. However, for very small r_{est} values (e.g. 10Mbps), the tail FCT (for the long flows) is much higher than FQ, due to the starvation effect explained before.

We try to capture this trade-off between isolation for short and medium sized flows and starvation for long flows, by using average FCT across bytes (in other words, the average FCT weighted by flow size) as our key metric. We term the r_{est} value that achieves the sweetest spot in this trade-off as the "best" r_{est} value. The r_{est} values that produce average FCT which is within 10% of the value produced by the best r_{est} are termed as "reasonable" r_{est} values. Table 2 presents our results across different settings. We find that (1) LSTF produces significantly lower average FCT than FIFO, performing only slightly worse than FQ (2) As expected, the best r_{est} value decreases with increasing utilization and with decreasing bandwidths (as in the case of I2 / 10 topology), while the range of reasonable r_{est} values gets narrower with increasing utilization and with decreasing bandwidths.

Thus, for instantaneous fairness, LSTF would require some estimate of the per-flow rate. We believe that this can be obtained from the knowledge of the network topology (in particular, the link bandwidths), which is available to the ISPs, and on-line measurement of traffic matrices and link utilization levels, which can be done using various tools [14, 18, 35]. However, this does impose a higher burden on deploying LSTF than on FQ or other such scheduling algorithms.

3.4 Limitations of LSTF: Policy-based objectives

So far we showed how LSTF achieves various performance objectives. We now describe certain policy-based objectives that are hard to achieve with LSTF.

Multi-tenancy: As network virtualization becomes more popular, networks are often called upon to support multiple tenants or traffic classes, with each having their own networking objectives. Network providers can enforce isolation across such tenants (or classes of traffic) through static bandwidth provisioning, which can be implemented via dedicated hard-wired links [1,5] or through multiqueue scheduling algorithms such as fair queuing or round robin [20]. LSTF can work in conjunction with both of these isolation mechanisms to meet different desired performance objectives for each tenant (or class of traffic).

However, without such multiqueue support it cannot provide such isolation or fairness on a per-class or per-tenant basis. This is because for class-based fairness (which also includes hierarchical fairness) the appropriate slack assignment for a packet at a particular ingress depends on the input from other ingresses (since these packets can belong to the same class). Note, however, that if two or more classes/tenants are separated by strict prioritization, LSTF can be used to enforce the appropriate precedence order, along with meeting the individual performance objective for each class.

Traffic Shaping: Shaping or rate limiting flows at a particular router requires non-work-conserving algorithms such as Token Bucket Filters [8]. LSTF itself is a work-conserving algorithm and cannot shape or rate limit the traffic on its own. We believe that shaping the traffic only at the edge, with the core remaining work-conserving, would also produce the desired network-wide behavior, though this requires further exploration.

4 Incorporating Network Feedback

Up until now we have considered packet scheduling in isolation, whereas in the Internet today routers send implicit feedback to hosts via packet drops [22, 32] (or marking, as in ECN [37]). This is often called Active Queue Management (AQM), and its goal is to reduce the per-packet delays while keeping throughput high. We now consider how we might generalize our LSTF approach to incorporate such network feedback as embodied in AQM schemes.

LSTF is just a scheduling algorithm and cannot perform AQM on its own. Thus, at first glance, one might think that incorporating AQM into LSTF would require implementing the AQM scheme in each router, which would then require us to find a universal AQM scheme in order to fulfill our pursuit of universality. On the contrary, LSTF enables a novel edge-based approach to AQM based on the following insights: (1) As long as appropriate packets are chosen, it does not matter *where* they are being dropped (or marked) – whether it is inside the core routers or at the edge. (2) In addition to scheduling packets LSTF produces a very useful by-product, carried by the slack values in the packets, which gives us a precise measure of the one-way queuing delay seen by the packet and can be used for AQM. For obtaining this by-product, an extra field is added to the packet header at the ingress which stores the assigned slack value (called the initial slack field), which remains untouched as the packet traverses the network. The other field where the ingress stores the assigned slack value is updated as per the LSTF algorithm; we call this the current slack field. The precise amount of queuing delay seen by the packet within the network (or the *used slack* value) can be computed at the edge by simply comparing the initial slack field and the current slack field.

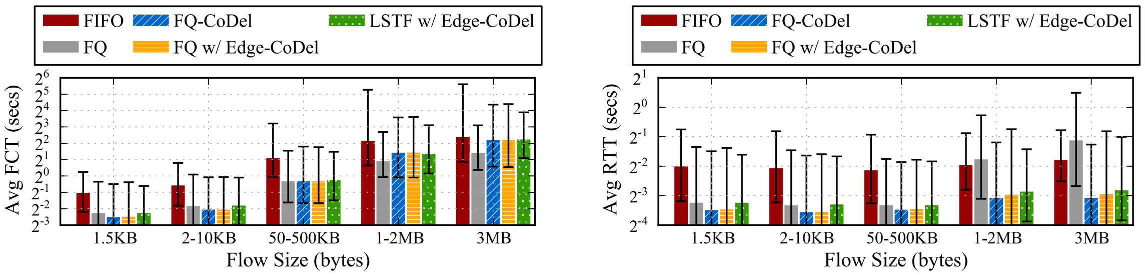
We evaluate our edge-based approach to AQM in the context of (1) CoDel [32], the state-of-the-art AQM scheme for wide area networks and (2) ECN used with DCTCP [9], the state-of-the-art AQM scheme for datacenters.

4.1 Emulating CoDel from Edge

Background: In CoDel, the amount of time a packet has spent in a queue is recorded as the sojourn time. A packet is dropped if its sojourn time exceeds a fixed target (set to 5ms [33]), and if the last packet drop happened beyond a certain interval (initialized to 100ms [33]). When a packet is dropped, the interval value is reduced using a control law, which divides the initial interval value by the square root of the number of packets dropped. The interval is refreshed (re-initialized to 100ms) when the queue becomes empty, or when a packet sees a sojourn time less than the target.[9] An extension to CoDel is FQ-CoDel [25], where the scheduler round-robins across flows and the CoDel control loop is applied to each flow individually. The interval for a flow is refreshed when there are no more packets belonging to that flow in the queue. FQ-CoDel is considered to be better than CoDel in all regards , even by one of the co-developers of CoDel [4].

Edge-CoDel: We aim to approximate FQ-CoDel from the edge by using LSTF to implement per-flow fairness in routers (as in §3.3). We then compute the used slack value at the egress router for every packet, as described above, and run the FQ-CoDel logic for when to drop packets for each flow, keeping the control law and the parameters (the target value and the initial interval value) the same as in

[9] CoDel is a little more complicated than this, and while our implementation follows the CoDel specification [33], our explanation has been simplified, highlighting only the relevant points for brevity.



Expt. Setup	r_{est}	Avg FCT across bytes (s)					Avg RTT across bytes (s)				
	(Mbps)	FIFO	FQ	FQ-CoDel	FQ w/ Edge-CoDel	LSTF w/ Edge-CoDel	FIFO	FQ	FQ-CoDel	FQ w/ Edge-CoDel	LSTF w/ Edge-CoDel
I2 1Gbps-10Gbps at 70% util.	100	0.811	0.622	0.642	0.633	0.641	0.0756	0.0733	0.0642	0.0646	0.0661
I2 1Gbps-1Gbps at 70% util.	100	0.766	0.630	0.642	0.637	0.658	0.0716	0.0702	0.0639	0.0643	0.0666
I2 / 10 at 30% util.	40	0.918	0.836	0.897	0.887	0.907	0.0998	0.1085	0.0792	0.0798	0.0826
I2 / 10 at 50% util.	30	1.706	1.214	1.430	1.369	1.427	0.1384	0.1752	0.0901	0.0918	0.1001
I2 / 10 at 70% util.	10	4.837	2.295	3.687	3.738	3.739	0.2779	0.3752	0.1182	0.1281	0.1388
I2 / 10, half RTTs at 70% util.	10	4.569	2.023	3.196	3.245	3.405	0.2555	0.3607	0.0995	0.1131	0.1165
I2 / 10, double RTTs at 70% util.	10	5.098	2.769	4.243	4.125	4.389	0.325	0.4172	0.1591	0.1640	0.1843
Rocketfuel at 70% util.	100	0.964	0.796	0.840	0.813	0.835	0.0922	0.0991	0.0794	0.0788	0.0836

Figure 6: The figures show the average FCT and RTT values for I2 / 10 at 70% utilization (LSTF uses fairness slack assignment with $r_{est} = 10Mbps$). The error bars indicate the 10^{th} and the 99^{th} percentile values and the y-axis is in log-scale. The table indicates the average FCT and RTTs (across bytes) for varying settings.

FQ-CoDel. We call this approach Edge-CoDel.

There are only two things that change in Edge-CoDel as compared to FQ-CoDel. First, instead of looking at the sojourn time of each queue individually, Edge-CoDel looks at the total queuing time of the packet across the entire network. The second change is with respect to how the CoDel interval is refreshed. As mentioned before, in traditional FQ-CoDel, there are two events that trigger a refresh in the interval (i) when a packet's sojourn time is less than the target and (ii) when all the queued-up packets for a given flow have been transmitted. While Edge-CoDel can react to the former, it has no explicit way of knowing the latter. To address this, we refresh the interval if the difference in the send time of two consecutive packets (found using TCP timestamps that are enabled by default) is more than a certain threshold. Clearly, this *refresh threshold* must be greater than CoDel's target queuing delay value. We find that a refresh threshold of 2-4 times the target value (10-20ms) works reasonably well.

Evaluation: In our experiments, we compare four different schemes: (1) FIFO without AQM (to set a baseline), (2) FQ without AQM (to see the effects of FQ on its own), (3) FQ-CoDel (to provide the state-of-the-art comparison) (4) LSTF scheduling (with slacks assigned to meet the fairness objective using appropriate r_{est} values) in conjunction with Edge-CoDel. As we move from (3) to (4), we make two transitions – first is with respect to the scheduling done inside the network (perfect isolation with FQ vs approximate isolation with LSTF) and the second is the shift of AQM logic from inside the network to the edge. Therefore, as an incremental step in between the two transitions, we also provide results for FQ with Edge-CoDel, where routers do FQ across flows (with the slack values maintained only for book-keeping) and AQM is done by Edge-CoDel. This allows us see how well Edge-CoDel works with perfect per-router isolation. The refresh threshold we use for Edge-CoDel in both cases is 20ms (4 times the CoDel target value). The buffer size is increased to 50MB so that AQM kicks in before a natural packet drop occurs.

Figure 6 shows our results for varying settings and schemes. The main metrics we use for evaluation are the FCTs and the per-packet RTTs, since the goal of an AQM scheme is to maintain high throughput (or small FCTs) while keeping the RTTs small. The two graphs show the average FCT and the average RTT across flows bucketed by their size for the I2 / 10 topology at 70% utilization (where AQM produces a bigger impact compared to our default case). As expected, we find that while FQ helps in reducing the FCT values as compared to FIFO, it results in significantly higher RTTs than FIFO for long flows. FQ-CoDel reduces the RTT seen by long flows compared to FQ (with the short flows having RTT smaller than FIFO and comparable to FQ). What is new is that, shifting the CoDel logic to the edge through Edge-CoDel while doing FQ in the router makes very little difference as compared to FQ-CoDel. As we experiment with varying settings, we find that in some cases, FQ with Edge-CoDel results in slightly smaller FCTs at the cost of slightly higher RTTs than FQ-CoDel. We believe that this is due to the difference in how the CoDel interval is refreshed with Edge-CoDel and with in-router FQ-CoDel. Replacing the scheduling algorithm with LSTF again produces minor differences in the results compared to FQ-CoDel. Both the FCT and the RTT are slightly higher than FQ-CoDel for almost all cases, and we attribute the differences to LSTF's *approximation* of round-robin service across flows. Nonetheless, the average FCTs obtained are significantly

Util.	Avg FCT (s)			Avg RTT (ms)		
	FIFO w/ No ECN	FIFO w/ ECN	LSTF w/ Edge-ECN	FIFO w/ No ECN	FIFO w/ ECN	LSTF w/ Edge-ECN
30%	0.0020	0.0011	0.0011	0.2069	0.1123	0.1077
50%	0.0219	0.0086	0.0079	0.3425	0.1601	0.1477
70%	0.0501	0.0241	0.0240	0.4497	0.2616	0.2494

Table 3: DCTCP performance with no ECN, ECN (in-switch) and Edge-ECN for the datacenter topology at varying utilizations.

lower than FIFO and the average RTTs are significantly lower than both FIFO and FQ for all cases.

Varying the refresh threshold used for Edge-CoDel produces minor differences in the aggregate results, a detailed evaluation of which can be found in Appendix D.

4.2 Emulating ECN for DCTCP from Edge

Background: DCTCP [9] is a congestion control scheme for datacenters that uses ECN marks as a congestion signal to control the queue size before a packet drop occurs. It requires the routers to mark the packets whenever the instantaneous queue size goes beyond a certain threshold K. These markings are echoed back to the sender with the acknowledgments and the sender decreases its sending rate in proportion to the ECN marked packets.

Edge-ECN: The marking process can be moved to the edge (or the receiving endhost) by simply marking a packet if its queuing delay (computed, as in §4.1, by subtracting the initial slack value from the current slack value) is greater than the transmission time of K packets. This transmission time is easy to compute in datacenters where the link capacities are known.

Evaluation: The results for varying utilization levels are shown in Table 3. We compare Edge-ECN running LSTF in the routers (with all packets initialized to the same slack value) with in-switch ECN running FIFO in the routers, both using the same unmodified DCTCP algorithm at the endhosts. We use the DCTCP default value of $K = 15$ packets as the marking threshold. We also present results for DCTCP with no ECN marks (which reduces to TCP) and FIFO scheduling, as a comparison point. We see that both in-switch ECN and Edge-ECN DCTCP have comparable performance, with significantly lower average FCTs and RTTs than no ECN TCP.

Summary: The *used slack* information available as a by-product from LSTF can be effectively used to emulate an AQM scheme from the edge of the network.

5 LSTF Implementation

In this section, we study the feasibility of implementing LSTF in the routers. We start with showing that given a switch that supports fine-grained priority scheduling, it is trivial to implement LSTF on it using programmable header processing mechanisms [14, 15]. We then explore two different proposals for implementing fine-grained priorities in hardware.

Using fine-grained priorities to implement LSTF: Consider a packet p that arrives at a router α at time $i(p,\alpha)$, with slack $slack(p,\alpha)$. As mentioned in §2, LSTF prioritizes packets based on their remaining slack value at the time when their last bit is transmitted. This term is given by $(slack(p,\alpha) - (t - i(p,\alpha)) + T(p,\alpha))$ at any time t while p is waiting at α. $T(p,\alpha)$ is the transmission time of p at α, which is added to account for the remaining slack of p, relative to other packets, when its *last bit* is transmitted. Since t is same for all packets at any given point of time when the packets are being compared at α, the deciding term is $(slack(p,\alpha) + i(p,\alpha) + T(p,\alpha))$. With $slack(p,\alpha)$ being available in the packet header and the values of $i(p,\alpha)$ and $T(p,\alpha)$ being available at α when the packet arrives at the router, this term can be easily computed and attached to the packet as its priority value. Right before a packet p is transmitted by the router, its slack can be overwritten by the remaining slack value, computed by subtracting the stored priority value $(slack(p,\alpha)+i(p,\alpha)+T(p,\alpha))$ with the sum of the current time and $T(p,\alpha)$. We verified that these steps can be easily executed using P4 [14].

Implementing fine-grained priorities in hardware: Fine-grained priorities can be implemented by using specialized data-structures such as pipelined heap (p-heap) [13, 26], which can scale to very large buffers ($>$100MB), because the pipeline stage time is not affected by the queue size. However, p-heaps are difficult to implement and verify due to their intricate design and large chip area, thus resulting in higher costs. The p-heap implemented by Ioannou et. al. [26] using a 130nm technology node has a per-port area overhead of 10% (over a typical switching chip with minimum area of $200mm^2$ [23]) [10].

Leveraging the advancement in hardware technology over the years, Sivaraman et. al. [41] propose a simpler solution, based on bucket-sort algorithm. The area overhead reduces to only 1.65% (over a baseline single-chip shared-memory switch such as the Broadcom Trident [2]), when implemented using a 16nm technology node. While this approach is much cheaper to implement, it cannot scale to very large buffer sizes (beyond a few tens of MBs).

Thus, given these choices, it does not appear a significant challenge to implement LSTF at linespeed, though the key trade-offs between cost, simplicity and buffer limits need to be taken into consideration. To support a scale-out infrastructure, most datacenters today use a large number of inexpensive single chip shared memory switches [40], which have shallow buffers (around 10MB). The low overhead bucket-sort based approach [41] towards implementing LSTF would be ideal in such a setup. Core routers in wide area, on the other hand, have deep buffers (a few hundred MBs) and would require the more expensive

[10] 130nm technology node was developed in 2001; the overheads would be lower for an implementation using the latest technology (14nm).

p-heap based implementation [13, 26]. While they are fewer in number [29], they may cost up to millions of dollars. Supporting the slightly more expensive, but flexible LSTF implementation would, to a large extent, obviate the need for replacing these expensive routers with changing demands, resulting in long-term savings. We are also optimistic that advancements in hardware technology would further reduce the cost overheads of implementing LSTF.

6 Related Work

The literature on packet scheduling is vast. Here we only touch on a few topics most relevant to our work.

The real-time scheduling literature has studied the optimality of scheduling algorithms[11] (in particular EDF and LSTF) for single and multiple processors [28, 30]. Liu and Layland [30] proved the optimality of EDF for a single processor in hard real-time systems. LSTF was then shown to be optimal for single-processor scheduling, while being more effective than EDF (though not optimal) for multi-processor scheduling [28]. In the context of networking, [17] provides theoretical results on emulating the schedules produced by a single output-queued switch using a combined input-output queued switch with a smaller speed-up of at most two. To the best of our knowledge, the optimality or universality of a scheduling algorithm for a network of inter-connected resources (in our case, switches) has never been studied before.

The authors of [42] propose the use of programmable hardware in the dataplane for packet scheduling and queue management, in order to achieve various objectives. The proposal shows that there is no "silver bullet" solution, by simulating three schemes (FQ, CoDel+FQ, CoDel+FIFO) competing on three different metrics. As mentioned earlier, our work is inspired by the questions the authors raise; we adopt a broader view of scheduling in which packets can carry dynamic state leading to the results presented here. A recent proposal for programmable packet scheduling [41], developed in parallel to UPS, uses an hierarchy of priority and calendar queues to express different scheduling algorithms on a single switch hardware. The proposed solution is able to achieve better expressiveness than LSTF by allowing packet headers to be re-initialized at *every switch*. UPS assumes a stronger model, where the header initialization is restricted to the ingress routers, while the core switches remain untouched. Moreover, we provide theoretical results which shed light on the effectiveness of both of these models.

7 Conclusion

This paper started with a theoretical perspective by analyzing whether there exists a single *universal* packet scheduling algorithm that can perfectly replay all viable schedules. We proved that while such an algorithm cannot exist, LSTF comes closest to being one (in terms of the number of congestion points it can handle). We then empirically demonstrated the ability of LSTF to approximately replay a wide range of scheduling algorithms under varying network settings. Replaying a given schedule, while of theoretical interest, requires the knowledge of viable output times, which is not available in practice.

Hence, we next considered if LSTF can be used in practice to achieve various performance objectives. We showed via simulation how LSTF, combined with heuristics to set the slack values at the ingress, can do a reasonable job of minimizing average flow completion time, minimizing tail latencies, and achieving per-flow fairness. We also discussed some limitations of LSTF (with respect to achieving class-based fairness and traffic shaping).

Noting that scheduling is often used along with AQM to prevent queue build up, we then showed how LSTF can be used to implement versions of AQM from the network edge, with performance comparable to FQ-CoDel and to DCTCP with ECN (the state-of-the art AQM schemes for wide-area and datacenters respectively).

While an initial step towards understanding the notion of a Universal Packet Scheduling algorithm, our work leaves several theoretical questions unanswered, three of which we mention here. First, we showed existence of a UPS with omniscient header initialization, and nonexistence with limited-information initialization. *What is the least information we can use for header initialization in order to achieve universality?* Second, we showed that, in practice, the fraction of overdue packets is small, and most are only overdue by a small amount. *Are there tractable bounds on both the number of overdue packets and/or their degree of lateness?* Third, while we have a formal characterization for the scope of LSTF with respect to replaying a given schedule, and we have simulation evidence of LSTF's ability to meet several performance objectives, we do not yet have any formal model for the scope of LSTF in meeting these objectives. *Can one describe the class of performance objectives that LSTF can meet?* Also, are there any new objectives that LSTF allows us to achieve?

8 Acknowledgments

We are thankful to Satish Rao for his helpful tips regarding the theoretical aspects of this work and to Anirudh Sivaraman for liberally sharing his insights on hardware implementation of fine-grained priorities. We would also like to thank Aisha Mushtaq, Kay Ousterhout, Aurojit Panda, Justine Sherry, Ion Stoica and our anonymous HotNets and NSDI reviewers for their thoughtful feedback. Finally, we would like to thank our shepherd Srikanth Kandula for helping shape the final version of this paper. This work was supported by Intel Research and by the National Science Foundation under Grant No. 1117161, 1343947 and 1040838.

[11] A scheduling algorithm is said to be optimal if it can (feasibly) schedule a set of tasks that can be scheduled by any other algorithm.

References

[1] Global Consortium to Construct New Cable System Linking US and Japan to Meet Increasing Bandwidth Demands. http://googlepress.blogspot.com/2008/02/global-consortium-to-construct-new_26.html.

[2] High Capacity StrataXGSTrident II Ethernet Switch Series. http://www.broadcom.com/products/Switching/Data-Center/BCM56850-Series.

[3] Internet2. http://www.internet2.edu/.

[4] Kathie Nichol's CoDel presented by Van Jacobson. http://www.ietf.org/proceedings/84/slides/slides-84-tsvarea-4.pdf.

[5] Microsoft Invests in Subsea Cables to Connect Datacenters Globally. http://goo.gl/GoXfxH.

[6] NS-2. http://www.isi.edu/nsnam/ns/.

[7] NS-3. http://www.nsnam.org/.

[8] Token Bucket Filters. http://lartc.org/manpages/tc-tbf.html.

[9] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proc. ACM SIGCOMM*, 2010.

[10] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *Proc. ACM SIGCOMM*, 2013.

[11] M. Allman. Comments on bufferbloat. *ACM SIGCOMM Computer Communication Review*, 2013.

[12] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. ACM Internet Measurement Conference (IMC)*, 2012.

[13] R. Bhagwan and B. Lin. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *Proc. IEEE Infocom*, 2000.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[15] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proc. ACM SIGCOMM*, 2013.

[16] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *Proc. ACM HotSDN*, 2012.

[17] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching output queueing with a combined input/output-queued switch. *IEEE Journal on Selected Areas in Communications*, 1999.

[18] B. Claise. Cisco systems NetFlow services export version 9. RFC 3954, 2004.

[19] D. D. Clark, S. Shenker, and L. Zhang. Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism. *ACM SIGCOMM Computer Communication Review*, 1992.

[20] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *ACM SIGCOMM Computer Communication Review*, 1989.

[21] N. Dukkipati and N. McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *ACM SIGCOMM Computer Communication Review*, 2006.

[22] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.*, 1993.

[23] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2013.

[24] A. Gupta, M. T. Hajiaghayi, and H. Räcke. Oblivious Network Design. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, 2006.

[25] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet. FlowQueue-Codel. *IETF Informational*, 2013.

[26] A. Ioannou and M. G. H. Katevenis. Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-speed Networks. *IEEE/ACM Trans. Netw.*, 2007.

[27] R. Jain, D.-M. Chiu, and W. Hawe. A Quantitative Measure Of Fairness And Discrimination For

Resource Allocation In Shared Computer Systems. *CoRR*, 1998.

[28] J. Y.-T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 1989.

[29] L. Li, D. Alderson, W. Willinger, and J. Doyle. A First-principles Approach to Understanding the Internet's Router-level Topology. In *Proc. ACM SIGCOMM*, 2004.

[30] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 1973.

[31] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively Cautious Congestion Control. In *Proc. USENIX NSDI*, 2014.

[32] K. Nichols and V. Jacobson. Controlling Queue Delay. *Queue*, 2012.

[33] K. Nichols and V. Jacobson. Controlled delay active queue management: draft-nichols-tsvwg-codel-02. *Internet Requests for Comments-Work in Progress, http://tools. ietf. org/id/draft-nichols-tsvwg-codel-01. txt, Tech. Rep*, 2014.

[34] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-node Case. *IEEE/ACM Trans. Netw.*, 1993.

[35] P. Phaal, S. Panchen, and N. McKee. InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks. RFC 3176, 2001.

[36] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-defined Internet Architecture: Decoupling Architecture from Infrastructure. In *Proc. ACM HotNets*, 2012.

[37] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP, 2001.

[38] S. Blake and D. Black and M. Carlson and E. Davies and Z. Wang and W. Weiss. An Architecture for Differentiated Services. RFC 2475, 1998.

[39] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. *ACM SIGCOMM Computer Communication Review*, 1995.

[40] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. ACM SIGCOMM*, 2015.

[41] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S.-T. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan. Towards Programmable Packet Scheduling. In *Proc. ACM HotNets*, 2015.

[42] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *Proc. ACM HotNets*, 2013.

[43] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. ACM SIGCOMM*, 2002.

[44] I. Stoica, S. Shenker, and H. Zhang. Core-stateless Fair Queueing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High-speed Networks. *IEEE/ACM Trans. Netw.*, 2003.

[45] I. Stoica and H. Zhang. Providing Guaranteed Services Without Per Flow Management. In *Proc. ACM SIGCOMM*, 1999.

[46] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM SIGCOMM Computer Communication Review*, 1990.

Appendix

A Proofs for Theoretical Results

This section contains theoretical proofs for the analytical replayability results presented in §2. We begin with defining some notations used throughout in the proofs.

A.1 Notations

We use the following notations for our proofs, some of which have been already defined in the main text:

Relevant nodes:
$src(p)$: Ingress of a packet p.
$dest(p)$: Egress of a packet p.

Relevant time notations:
$T(p,\alpha)$: Transmission time of a packet p at node α.
$o(p,\alpha)$: Time when the first bit of p is scheduled by node α in the original schedule.
$o(p) = o(p,dest(p)) + T(p,dest(p))$: Time when the last bit of p exits the network in the original schedule (which is non-preemptive).
$o'(p)$: Time when the last bit of p exits the network in the replay (which may be preemptive in our theoretical arguments).
$i(p,\alpha)$ and $i'(p,\alpha)$: Time when p arrives at node α in the original schedule and in the replay respectively.

$i(p) = i(p,src(p)) = i'(p)$: Arrival time of p at its ingress. This remains the same for both the original schedule and the replay.
$t_{min}(p,\alpha,\beta)$: Minimum time p takes to start from node α and exit from node β in an uncongested network. It therefore includes the propagation delays and the store-and-forward delays of all links in the path from α to β and the transmission delays at α and β. Handling the edge case: $t_{min}(p,\alpha,\alpha) = T(p,\alpha)$
$slack(p) = o(p) - i(p) - t_{min}(p,src(p),dest(p))$: Total slack of p that gets assigned at its ingress. It denotes the amount of time p can wait in the network (excluding the time when any of its bits are getting serviced) without missing its target output time.
$slack(p,\alpha,t) = o(p) - t - t_{min}(p,\alpha,dest(p)) + T(p,\alpha)$: Remaining slack of the last bit of p at time t when it is at node α. We derive this expression in Appendix A.4.

Other miscellaneous notations:
$path(p,\alpha,\beta)$: The ordered set of nodes and links in the path taken by p to go from α to β. The set also includes α and β as the first and the last nodes.
$path(p) = path(p,src(p),dest(p))$
$pass(\alpha)$: Set of packets that pass through node α.

A.2 Existence of a UPS under Omniscient Header Initialization

Algorithm: At the ingress, insert an n-dimensional vector in the packet header, where the i^{th} element contains $o(p,\alpha_i)$, α_i being the i^{th} hop in $path(p)$. Every time a packet p arrives at the router, the router pops the value at the head of the vector in p's header and uses that as the priority for p (earlier values of output times get higher priority). This can perfectly replay any schedule.

Proof: We can prove that the above algorithm will result in no overdue packets (which do not meet their original schedule's target) using the following two theorems:
Theorem 1: If for any node α, $\exists p' \in pass(\alpha)$, such that using the above algorithm, the last bit of p' exits α at time $(t' > (o(p',\alpha) + T(p',\alpha)))$, then $(\exists p \in pass(\alpha) \,|\, i'(p,\alpha) \leq t'$ and $i'(p,\alpha) > o(p,\alpha))$.
Proof by contradiction: Consider the first such $p^* \in pass(\alpha)$ that gets late at α (i.e. its last bit exits α at time $t^* > (o(p^*,\alpha) + T(p^*,\alpha))$). Suppose the above condition is not true i.e. $(\forall p \in pass(\alpha) \,|\, i'(p,\alpha) \leq o(p,\alpha)$ or $i'(p,\alpha) > t^*)$. In other words, if p arrives at or before time t^*, it also arrives at or before time $o(p,\alpha)$. Given that all bits of p^* arrive at or before time t^*, they also arrive at or before time $o(p^*,\alpha)$. The only reason why the last bit of p^* would wait until time $(t^* > o(p^*,\alpha) + T(p^*,\alpha))$ in our work-conserving replay is if some other bits (belonging to higher priority packets) were being scheduled after time $o(p^*,\alpha)$, resulting in p^* not being able to complete its transmission by time $(o(p^*,\alpha) + T(p^*,\alpha))$. However, as per our algorithm, any packet p_{high} having a higher priority than p^* at α must have been scheduled before p^* in the original schedule, implying that $(o(p_{high},\alpha) + T(p_{high},\alpha)) \leq o(p^*,\alpha)$.[12] Therefore, some bits of p_{high} being scheduled after time $o(p^*,\alpha)$, implies them being scheduled after time $(o(p_{high},\alpha) + T(p_{high},\alpha))$. This means that p_{high} is already late and contradicts our assumption that p^* is the first packet to get late. . Hence, Theorem 1 is proved by contradiction.
Theorem 2: $\forall \alpha, (\forall p \in pass(\alpha) \,|\, i'(p,\alpha) \leq i(p,\alpha))$.
Proof by contradiction: Consider the first time when some packet p^* arrives late at some node α^* (i.e. $i'(p^*,\alpha^*) > i(p^*,\alpha^*)$). In other words, α^* is the first node in the network to see a late packet arrival, and p^* is the first late arriving packet. Let α_{prev} be the node visited by p^* just before arriving at α^*. p^* can arrive at a time later than $i(p^*,\alpha^*)$ at α^* only if the last bit of p^* exits α_{prev} at time $t_{prev} > o(p^*,\alpha_{prev}) + T(p^*,\alpha_{prev})$. As per Theorem 1 above, this is possible only if some packet p' (which may or may not be the same as p^*) arrives at α_{prev} at time $i'(p',\alpha_{prev}) > o(p',\alpha_{prev}) \geq i(p',\alpha_{prev})$ and $i'(p',\alpha_{prev}) \leq t_{prev} < i'(p^*,\alpha^*)$. This contradicts our assumption that α^* is the first node to see a late arriving packet. Therefore, $\forall \alpha, (\forall p \in pass(\alpha) \,|\, i'(p,\alpha) \leq i(p,\alpha))$.

Combining the two theorems above: Since $\forall \alpha (\forall p \in pass(\alpha) \mid i'(p,\alpha) \leq i(p,\alpha))$, with the above algorithm, $\forall \alpha (\forall p \in pass(\alpha))$, all bits of p exit α before $(o(p,\alpha) + T(p,\alpha))$. Therefore, the algorithm can perfectly replay any viable schedule.

A.3 Nonexistence of a UPS under black-box initialization

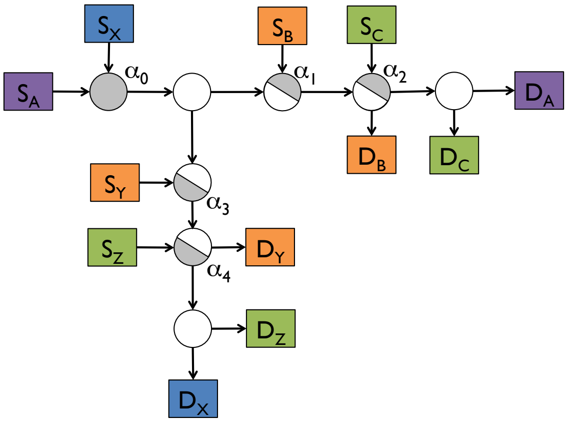
Proof by counter-example: Consider the example shown in Figure 7. For simplicity, assume all the propagation delays are zero, the transmission time for each congestion point (shaded in gray) is 1 unit and the uncongested (white) routers have zero transmission time.[13] All packets are of the same size.

The table illustrates two cases. For each case, a packet's arrival and scheduling time (the time when the packet is scheduled by the router) at each node through which it passes are listed. A packet represented by p belongs to flow P, with ingress S_P and egress D_P, where $P \in \{A,B,C,X,Y,Z\}$. The packets have the same *path* in both cases. For example, a belongs to Flow A, starts at ingress S_A, exits at egress D_A and passes through three congestion points in its path α_0, α_1 and α_2; x belongs to Flow X, starts at ingress S_X, exits at egress D_X and passes through three congestion points in its path α_0, α_3 and α_4; and so on.

The two critical packets we care about in this example

[12] Given that the original schedule is non-preemptible, the next packet gets scheduled only after the previous one has completed its transmission.

[13] These assignments are made for simplicity of understanding. The example will hold for any reasonable value of propagation and transmission delays.



Node	Packet(arrival time, scheduling time)
	Case 1
α_0	$a(\mathbf{0},0); x(\mathbf{0},1)$
α_1	$a(1,1), b_1(2,2), b_2(3,3), b_3(4,4)$
α_2	$c_1(2,2), c_2(3,3); a(2,\mathbf{4})$
α_3	$x(2,2), y_1(2,3), y_2(3,4)$
α_4	$z(2,2), x(3,\mathbf{3})$
	Case 2
α_0	$x(\mathbf{0},0); a(\mathbf{0},1)$
α_1	$a(2,2), b_1(2,3), b_2(3,4), b_3(4,5)$
α_2	$c_1(2,2), c_2(3,3), a(3,\mathbf{4})$
α_3	$x(1,1), y_1(2,2), y_2(3,3)$
α_4	$z(2,2), x(2,\mathbf{3})$

Figure 7: Example showing non-existence of a UPS with Blackbox Initialization. A packet represented by p belongs to flow P, with ingress S_P and egress D_P, where $P \in \{A,B,C,X,Y,Z\}$. For simplicity assume all packets are of the same size and all links have a propagation delay of zero. All uncongested routers (white), ingresses and egresses have a transmission time of zero. The congestion points (shaded gray) have transmission times of $T = 1$ unit.

are a and x, which interact with each-other at their first congestion point α_0, being scheduled by α_0 at different times in the two cases (a before x in Case 1 and x before a in Case 2). But, notice that for both cases,

1. a enters the network from its ingress S_A at congestion point α_0 at time 0, and passes through two other congestion points α_1 and α_2 before exiting the network at time $(4+1)$ [14].
2. x enters the network from its ingress S_X at congestion point α_0 at time 0, and passes through two other congestion points α_3 and α_4 before exiting the network at time $(3+1)$.

a interacts with packets from Flow C at its third congestion point α_2, while x interacts with a packet from Flow Z at its third congestion point α_4. For both cases,

1. Two packets of Flow C (c_1, c_2) enter the network at times 2 and 3 at α_2 before they exit the network at time $(2+1)$ and $(3+1)$ respectively.
2. z enters the network at time 2 at α_4 before exiting at time $2+1$.

The difference between the two cases comes from how a interacts with packets from Flow B at its second congestion point α_1 and how x interacts with packets from Flow Y at its second congestion points α_3. Note that α_1 and α_3 are the last congestion points for Flow B and Flow Y packets respectively and their exit times from these congestion points directly determine their exit times from the network.

1. Three packets of Flow B (b_1,b_2,b_3) enter the network at times 2, 3 and 4 respectively at α_1. In Case 1, they leave α_1 at times $(2+1),(3+1),(4+1)$ respectively. This provides no *lee-way* for a at α_0, which leaves α_1 at time $(1+1)$, since it is required that α_1 must schedule a by at most time 3 in order for it to exit the network at its target output time. In Case 2, (b_1,b_2,b_3) leave at times $(3+1),(4+1),(5+1)$ respectively, providing lee-way for a at α_0, which leaves α_1 at time $(2+1)$.
2. Two packets of Flow Y (y_1,y_2) enter the network at times 2 and 3 respectively at α_3. In Case 1, they leave at times $(3+1),(4+1)$ respectively, providing a lee-way for x at α_0, which leaves α_3 at time $(2+1)$. In Case 2, (y_1,y_2) exit at times $(2+1),(3+1)$, providing no lee-way for x at α_0, which leaves α_3 at time $(1+1)$.

Note that the interaction of a and x with Flow C and Flow Z at their third congestion points respectively, is what ensures that their eventual exit time remains the same across the two cases inspite of the differences in how a and x are scheduled in their previous two hops.

Thus, we can see that $i(a)$, $o(a)$, $i(x)$, $o(x)$ are the same in both cases (also indicated in bold blue). Yet, *Case 1* requires a to be scheduled before x at α_0, else packets will get delayed at α_1, since it is required that α_1 schedules a at a time no more than 3 units if it is to meet its target output time. *Case 2* requires x to be scheduled before a at α_0, else packets will be delayed at α_3, where it is required to schedule x at a time no more than 2 units if it is to meet its target output time. Since the attributes $(i(\cdot),o(\cdot),path(\cdot))$ for both a and x are exactly the same in both cases, any deterministic UPS with Blackbox Initialization will produce the same order for the two packets at α_0, which contradicts the situation where we want a before x in one case and x before a in another.

A.4 Deriving the Slack Equation

We now prove that for any packet p waiting at any node α at time t_{now}, the remaining slack of the last bit of p is given by $slack(p, \alpha, t_{now}) = o(p) - t_{now} - t_{min}(p,\alpha,dest(p)) + T(p,\alpha)$.

Let $t_{wait}(p,\alpha,t_{now})$ denote the total time spent by p on waiting behind other packets at the nodes in its path from $src(p)$ to α (including these two nodes) until time t_{now}. We define $t_{wait}(p,\alpha,t_{now})$, such that it excludes the transmission times at previous nodes which gets captured in t_{min}, but includes the local service time received by the

[14] $+1$ is added to indicate transmission time at the last congestion point. As mentioned before, we assume the propagation delay to the egress and the transmission time at the egress are both 0.

packet so far at α itself.

$$\begin{aligned}
slack(p,\alpha,t_{now}) &= slack(p) - t_{wait}(p,\alpha,t_{now}) + T(p,\alpha) && \text{(1a)}\\
&= o(p) - i(p) - t_{min}(p,src(p),dest(p)) \\
&\quad - t_{wait}(p,\alpha,t_{now}) + T(p,\alpha) && \text{(1b)}\\
&= o(p) - i(p) - (t_{min}(p,src(p),\alpha) \\
&\quad + t_{min}(p,\alpha,dest(p)) - T(p,\alpha)) \\
&\quad - t_{wait}(p,\alpha,t_{now}) + T(p,\alpha) && \text{(1c)}\\
&= o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha) \\
&\quad - (i(p) + t_{min}(p,src(p),\alpha) \\
&\quad - T(p,\alpha) + t_{wait}(p,\alpha,t_{now})) && \text{(1d)}\\
&= o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha) - t_{now} && \text{(1e)}
\end{aligned}$$

(1a) is straightforward from our definition of LSTF and how the slack gets updated at every time slice. $T(p,\alpha)$ is added since α needs to locally consider the slack of the last bit of the packet in a store-and-forward network. (1c) then uses the fact that for any α in $path(p)$, $(t_{min}(p,src(p),dest(p)) = t_{min}(p,src(p),\alpha) + t_{min}(p,\alpha,dest(p)) - T(p,\alpha))$. $T(p,\alpha)$ is subtracted here as it is accounted for twice when we break up the equation for $t_{min}(p,src(p),dest(p))$. (1e) then follows from the fact that the difference between t_{now} and $i(p)$ is equal to the total amount of time the packet has spent in the network until time t_{now} i.e. $(t_{now} - i(p) = (t_{min}(p,src(p),\alpha) - T(p,\alpha)) + t_{wait}(p,\alpha,t_{now}))$. We need to subtract $T(p,\alpha)$, since by our definition, $t_{min}(p,src(p),\alpha)$ includes transmission time of the packet at α.

A.5 LSTF and EDF Equivalence

In our network-wide extension of EDF scheduling, every router computes a deadline (or priority) for a packet p based on the static header value $o(p)$ and additional state information about the minimum time the packet would take to reach its destination from the router. More precisely, each router (say α), uses $priority(p) = (o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha))$ to do priority scheduling, with $o(p)$ being the value carried by the packet header, initialized at the ingress and remaining unchanged throughout. EDF is equivalent to LSTF, in that for a given original schedule, the two produce exactly the same replay schedule.

Proof: Consider a node α and let $P(\alpha,t_{now})$ be the set of packets waiting at the output queue of α at time t_{now}. A packet will then be scheduled by α as follows:
With EDF: Schedule packet $p_{edf}(\alpha,t_{now})$, where

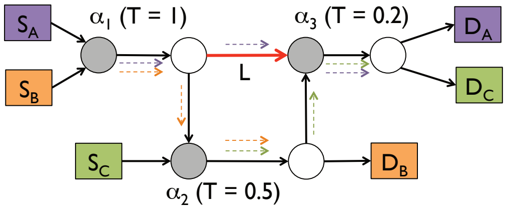
$$p_{edf}(\alpha,t_{now}) = \underset{p\in P(\alpha,t_{now})}{\operatorname{argmin}} \; (priority(p,\alpha))$$

$$priority(p,\alpha) = o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha)$$

With LSTF: Schedule packet $p_{lstf}(\alpha,t_{now})$, where

$$p_{lstf}(\alpha,t_{now}) = \underset{p\in P(\alpha,t_{now})}{\operatorname{argmin}} \; (slack(p,\alpha,t_{now}))$$

$$slack(p,\alpha,t_{now}) = o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha) - t_{now}$$



Node	Packet(arrival time, scheduling time)
α_1	$a(0,0),b(0,1)$
α_2	$b(2,2),c(2,2.5)$
α_3	$c(3,3),a(3,3.2)$

Figure 8: Example showing replay failure with simple priorities for a schedule with two congestion points per packet. A packet represented by p belongs to flow P, with ingress S_P and egress D_P, where $P \in \{A,B,C\}$. All packets are of the same size. For simplicity assume all links (except L) have a propagation delay of zero. L has a propagation delay of 2. All uncongested routers (white circles), ingresses and egresses have a transmission time of zero. The three congestion points – $\alpha_1,\alpha_2,\alpha_3$ have transmission times of $T=1$ unit, $T=0.5$ units and $T=0.2$ units respectively.

The above expression for $slack(p,\ \alpha,\ t_{now})$ has been derived in §A.4. Thus, $slack(p,\ \alpha,\ t_{now}) = priority(p,\alpha) - t_{now}$. Since t_{now} is the same for all packets, we can conclude that:

$$\underset{p\in P(\alpha,t_{now})}{\operatorname{argmin}} \; (slack(p,\alpha,t_{now})) = \underset{p\in P(\alpha,t_{now})}{\operatorname{argmin}} \; (priority(p,\alpha))$$

$$\implies p_{lstf}(\alpha,t_{now}) = p_{edf}(\alpha,t_{now})$$

Therefore, at any given point of time, all nodes will schedule the same packet with both EDF and LSTF (assuming ties are broken in the same way for both EDF and LSTF, such as by using FCFS). Hence, EDF and LSTF are equivalent.

A.6 Simple Priorities Replay Failure for Two Congestion Points Per Packet

In Figure 8, we present an example which shows that simple priorities can fail in replay when there are two congestion points per packet, no matter what information is used to assign priorities. At α_1, we need to have $priority(a) < priority(b)$, at α_2 we need to have $priority(b) < priority(c)$ and at α_3 we need to have $priority(c) < priority(a)$. This creates a priority cycle where we need $priority(a) < priority(b) < priority(c) < priority(a)$, which can never be possible to achieve with simple priorities.

We would also like to point out here that priority assignment for perfect replay in networks with single congestion point per packet requires detailed knowledge about the topology and the input load. More precisely, if a packet p passes through congestion point α_p, then its priority needs to be assigned as $priority(p) = o(p) - t_{min}(p,\alpha_p,dest(p)) + T(p,\alpha_p)$. The proof that this would always replay schedules with at most one congestion point per packet follows from the fact that the only scheduling decision made in a packet p's path is at the single congestion point α_p. This decision, at the single congestion point in a packet's path, is the same as what will be made with the network-wide extension of EDF, which we proved is equivalent to LSTF in §A.5. LSTF, in turn, can always replay schedules with one (or to be more precise, at most two) congestion points per packet, as we shall prove in §A.7.

Hence, in order to replay schedules with at most one congestion per packet using simple priorities, we need to know where the congestion point occurs in a packet's path, along with the final output times, to assign the priorities. In the absence of this knowledge, priorities cannot replay even a single congestion point.

A.7 LSTF: Perfect Replay for at most Two Congestion Points per Packet

A.7.1 Main Proof

We now prove that LSTF can replay all schedules with at most two congestion points per packet. Note that we work with bits in our proof, since we assume a preemptive version of LSTF. Due to store-and-forward routers, the remaining slack of a packet at a particular router is represented by the slack of the last bit of the packet (with all other bits of the packet having the same slack as the last bit).

In order for a replay failure to occur, there must be at least one overdue packet, where a packet p is said to be overdue if $o'(p) > o(p)$. This implies that p must have spent all of its slack while waiting behind other packets at a queue in some node α at say time t, such that $slack(p,\alpha,t) < 0$. Obviously, α must be a congestion point.

Necessary Condition for Replay Failure with LSTF: If a packet p^* sees negative slack at a congestion point α when its last bit exits α at time t^* in the replay (i.e. $slack(p^*,\alpha,t^*) < 0$), then $(\exists p \in pass(\alpha) \mid i'(p,\alpha) \leq t^*$ and $i'(p,\alpha) > o(p,\alpha))$. We prove this in §A.7.2.

We use the term "*local deadline* of p at α" for $o(p,\alpha)$, which is the time at which α schedules p in the original schedule.

Key Observation: *When there are at most two congestion points per packet, then no packet p can arrive at any congestion point α in the replay, after its local deadline at α (.i.e. $i'(p,\alpha) > o(p,\alpha)$ is not possible). Therefore, by the necessary condition above, no packet can see a negative slack at any congestion point.*

Proof by contradiction: Suppose that there exists α^*, which is the first congestion point (in time) that sees a packet which arrives after its local deadline at α^*. Let p^* be this first packet that arrives after its local deadline at α^* ($i'(p^*,\alpha^*) > o(p^*,\alpha^*)$). Since there are at most two congestion points per packet, either α^* is the first congestion point seen by p^* or the last (or both).

(1) If α^* is the first congestion point seen by p^*, then clearly, $i'(p^*,\alpha^*) = i(p^*,\alpha^*) \leq o(p^*,\alpha^*)$. This contradicts our assumption that $i'(p^*,\alpha^*) > o(p^*,\alpha^*)$.

(2) If α^* is not the first congestion point seen by p^*, then it is the last congestion point seen by p^*. If $i'(p^*,\alpha^*) > o(p^*,\alpha^*)$, then it would imply that p^* saw a negative slack before arriving at α^*. Suppose p^* saw a negative slack at a congestion point α_{prev}, before arriving at α^* when its last bit exited α_{prev} at time t_{prev}. Clearly, $t_{prev} < i'(p^*,\alpha^*)$. As per our necessary condition, this would imply that there must be another packet p', such that $i'(p',\alpha_{prev}) > o(p',\alpha_{prev})$ and $i'(p',\alpha_{prev}) \leq t_{prev} < i'(p^*,\alpha^*)$. This contradicts our assumption that α^* is the first congestion point (in time) that sees a packet which arrives after its corresponding scheduling time in the original schedule.

Hence, no congestion point can see a packet that arrives after its local deadline at that congestion point (and therefore no packet can get overdue) when there are at most two congestion points per packet.

A.7.2 Proof for Necessary Condition for Replay Failure with LSTF

We start this proof with the following observation:

Observation 1: If all bits of a packet p exit a router α by time $o(p,\alpha)+T(p,\alpha)$, then p cannot see a negative slack at α.

Proof for Observation 1: As shown previously in §A.4,

$$slack(p,\alpha,t) = o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha) - t$$

Therefore,

$$slack(p,\alpha,o(p,\alpha)+T(p,\alpha))$$
$$= o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha) - (o(p,\alpha)+T(p,\alpha))$$
$$\text{But, } o(p) = o(p,\alpha) + t_{min}(p,\alpha,dest(p)) + wait(p,\alpha,dest(p))$$
$$\implies slack(p,\alpha,o(p,\alpha)+T(p,\alpha)) = wait(p,\alpha,dest(p))$$
$$\implies slack(p,\alpha,o(p,\alpha)+T(p,\alpha)) \geq 0$$

where $wait(p,\alpha,dest(p))$ is the time spent by p in waiting behind other packets in the original schedule, after it left α, which is clearly non-negative.

We now move to the main proof for the necessary condition.

Necessary Condition for Replay Failure: If a packet p^* sees negative slack at a congestion point α when its last bit exits α at time t^* in the replay (i.e. $slack(p^*,\alpha,t^*) < 0$), then $(\exists p \in pass(\alpha) \mid i'(p,\alpha) \leq t^*$ and $i'(p,\alpha) > o(p,\alpha))$.

Proof by Contradiction: Suppose this is not the case .i.e. there exists p^* whose last bit exits α at time t^*, such that $slack(p^*,\alpha,t^*) < 0$ and $(\forall p \in pass(\alpha) \mid i'(p,\alpha) > t^*$ or $i'(p,\alpha) \leq o(p,\alpha))$. We can show that if the latter condition holds, then p^* cannot see a negative slack at α, thus violating our assumption.

We take the set of all bits which exit α at or before time t^* in the LSTF replay schedule. We denote this set as $S_{bits}(\alpha,t^*)$. As per our assumption, $(\forall b \in S_{bits}(\alpha,t^*) \mid i'(p_b,\alpha) \leq o(p_b,\alpha))$, where p_b denotes the packet to which bit b belongs. Note that $S_{bits}(\alpha,t^*)$ also includes all bits of p^*, since they all arrive before time t^*.

We now prove that no bit in $S_{bits}(\alpha,t^*)$ can see a negative slack (and therefore p^* cannot see a negative slack at α), leading to a contradiction. The proof comprises of two steps:

Step 1: Using the same input arrival times of each packet at α as in the replay schedule, we first construct a *feasible schedule* at α up until time t^*, denoted by $FS(\alpha,t^*)$, where by feasibility we mean that no bit in $S_{bits}(\alpha,t^*)$ sees a negative slack.

Step 2: We then do an iterative transformation of $FS(\alpha,t^*)$ such that the bits in $S_{bits}(\alpha,t^*)$ are scheduled in the order of their *least remaining slack times*. This reproduces the LSTF replay schedule from which $FS(\alpha,t^*)$ was constructed in the first place. However, while doing the transformation we show how the schedule remains feasible at every iteration, proving that the LSTF schedule finally obtained is also feasible up until time t^*. In other words, no packet sees a negative slack at α in the resulting LSTF replay schedule up until time t^*, contradicting our assumption that p^* sees a negative slack when it exits α at time t^* in the replay.

We now discuss these two steps in details.

Step 1: Construct a feasible schedule at α up until time t^* (denoted as $FS(\alpha,t^*)$) for which no bit in $S_{bits}(\alpha,t^*)$ sees a negative slack.

(i) Algorithm for constructing $FS(\alpha,t^*)$: Use priorities to schedule each bit in $S_{bits}(\alpha,t^*)$, where $\forall b \in S_{bits}(\alpha,t^*) \mid priority(b) = o(p_b,\alpha)$. (Note that since both $FS(\alpha,t^*)$ and LSTF are work-conserving, $FS(\alpha,t^*)$ is just a shuffle of the LSTF schedule up until t^*. The set of time slices at which a bit is scheduled in $FS(\alpha,t^*)$ and in the LSTF schedule up until t^* remains the same, but *which* bit gets scheduled at a given time slice is different.)

(ii) In $FS(\alpha,t^*)$, all bits b in $S_{bits}(\alpha,t^*)$ exit α by time $o(p_b,\alpha)+T(p_b,\alpha)$.

Proof by contradiction: Suppose the statement is not true and consider the first bit b^* that exits after time $(o(p_{b^*},\alpha)+T(p_{b^*},\alpha))$. We term this as b^* got late at α due to $FS(\alpha,t^*)$. Remember that, as per our assumption, $(\forall b \in S_{bits}(\alpha,t^*) \mid i'(p_b,\alpha) \le o(p_b,\alpha))$. Thus, given that all bits of p_{b^*} arrive at or before time $o(p_{b^*},\alpha)$, the only reason why the delay can happen in our work-conserving $FS(\alpha,t^*)$ is if some other higher priority bits were being scheduled after time $o(p_{b^*},\alpha)$, resulting in p_{b^*} not being able to complete its transmission by time $(o(p_{b^*},\alpha)+T(p_{b^*},\alpha))$. However, as per our priority assignment algorithm, any bit b' having a higher priority than b^* at α must have been scheduled before the first bit of p_{b^*} in the non-preemptible original schedule, implying that $(o(p_{b'},\alpha)+T(p_{b'},\alpha)) \le o(p_{b^*},\alpha)$. Therefore, a bit b' being scheduled after time $o(p_{b^*},\alpha)$, implies it being scheduled after time $(o(p_{b'},\alpha)+T(p_{b'},\alpha))$. This contradicts our assumption that b^* is the first bit to get late at α due to $FS(\alpha,t^*)$. Therefore, all bits b in $S_{bits}(\alpha,t^*)$ exit α by time $o(p_b,\alpha)+T(p_b,\alpha)$ as per the schedule $FS(\alpha,t^*)$.

(iii) Since all bits in $S_{bits}(\alpha,t^*)$ exit by time $o(p_b,\alpha)+T(p_b,\alpha)$ due to $FS(\alpha,t^*)$, no bit in $S_{bits}(\alpha,t^*)$ sees a negative slack at α (from Observation 1).

Step 2: *Transform* $FS(\alpha, t^*)$ into a feasible LSTF schedule for the single switch α up until time t^*.

(Note: The following proof is inspired from the standard LSTF optimality proof that shows that for a single switch, any feasible schedule can be transformed to an LSTF (or EDF) schedule [30]).

Let $fs(b,\alpha,t^*)$ be the scheduling time slice for bit b in $FS(\alpha,t^*)$. The transformation to LSTF is carried out by the following pseudo-code:

```
1: while true do
2:     Find two bits, $b_1$ and $b_2$, such that:
           $(fs(b_1,\alpha,t^*) < fs(b_2,\alpha,t^*))$ and
           $(slack(b_2,\alpha,fs(b_1,\alpha,t^*))$
             $< slack(b_1,\alpha,fs(b_1,\alpha,t^*)))$ and
           $(i'(b_2,\alpha,t^*) \le fs(b_1,\alpha,t^*))$
3:     if no such $b_1$ and $b_2$ exist then
4:         $FS(\alpha,t^*)$ is an LSTF schedule
5:         break
6:     else
7:         $swap(fs(b_1,\alpha,t^*),fs(b_2,\alpha,t^*))$    ▷ swap the scheduling times of the two bits. [15]
8:     end if
9: end while
10: Shuffle the scheduling time of the bits belonging to the same packet, to ensure that they are in order.
11: Shuffle the scheduling time of the same-slack bits such that they are in FIFO order
```

Line 7 above will not cause b_1 to have a negative slack, when it gets scheduled at $fs(b_2,\alpha,t^*)$ instead of $fs(b_1,\alpha,t^*)$. This is because the difference in $slack(b_2,\alpha,t)$ and $slack(b_1,\alpha,t)$ is independent of t and so:

$$slack(b_2,\alpha,fs(b_1,\alpha,t^*)) < slack(b_1,\alpha,fs(b_1,\alpha,t^*))$$
$$\implies slack(b_2,\alpha,fs(b_2,\alpha,t^*)) < slack(b_1,\alpha,fs(b_2,\alpha,t^*))$$

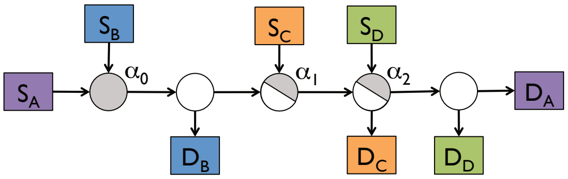
Since $FS(\alpha, t^*)$ is feasible before the swap, $slack(b_2, \alpha, fs(b_2, \alpha, t^*)) \ge 0$. Therefore, $slack(b_1,\alpha,fs(b_2,\alpha,t^*)) > 0$ and the resulting $FS(\alpha,t^*)$ after the swap remains feasible.

Lines 10 and 11 will also not result in any bit getting a negative slack, because all bits participating in the shuffle have the same slack at any fixed point of time in α.

Therefore, no bit in $S_{bits}(\alpha,t^*)$ has a negative slack at α after any iteration.

Since no bit in $S_{bits}(\alpha,t^*)$ has a negative slack at α in the swapped LSTF schedule, it contradicts our statement

[15] Note that we are working with bits here for easy expressibility. In practice, such a swap is possible under the preemptive LSTF model.



Original Schedule	
Node	*Packet(arrival time, scheduling time)*
α_0	$a(0,0), b(0,1)$
α_1	$a(1,1), c_1(2,2), c_2(3,3)$
α_2	$d_1(2,2), d_2(3,3), a(2,4)$
LSTF Replay	
Node	*Packet(arrival time, scheduling time)*
α_0	$b(0,0), a(0,1)$
α_1	$c_1(2,2), a(2,3), c_2(3,4)$
α_2	$d_1(2,2), d_2(3,3), a(4,4)$

Figure 9: Example showing replay failure with LSTF when there is a flow with three congestion points. A packet represented by p belongs to flow P, with ingress S_P and egress D_P, where $P \in \{A, B, C, D\}$. For simplicity assume all links have a propagation delay of zero. All uncongested routers (white), ingresses and egresses have a transmission time of zero. The three congestion points (shaded gray) have transmission times of $T = 1$ unit

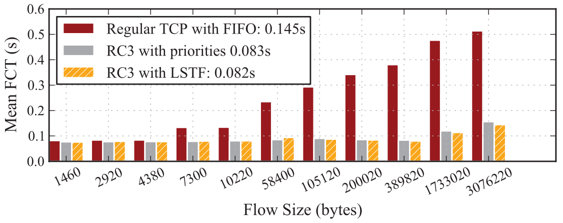
that p^* sees a negative slack when its last bit exits α at time t^*. Hence proved that if a packet p^* sees a negative slack at congestion point α when its last bit exits α at time t^* in the replay, then there must be at least one packet that arrives at α in the replay at or before time t^* and later than the time at which it is scheduled by α in the original schedule.

A.7.3 Replay Failure Example with LSTF

In Figure 9, we present an example where a flow passes through three congestion points and a replay failure occurs with LSTF. When packet a arrives at α_0, it has a slack of 2 (since it waits behind d_1 and d_2 at α_2), while at the same time, packet b has a slack of 1 (since it waits behind a at α_0). As a result, b gets scheduled before a in the LSTF replay. a therefore arrives at α_1 with slack 1 at time 2. c_1 with a zero slack is prioritized over a. This reduces a's slack to zero at time 3, when c_2 is also present at α_1 with zero slack. Scheduling a before c_2, will result in c_2 being overdue (as shown). Likewise, scheduling c_2 before a would have resulted in a getting overdue. Note that in this failure case, a arrives at α_1 at time 2, which is greater than $o(a, \alpha_1) = 1$.

B Minimizing Average FCT by using RC3 with LSTF

We now look at how in-network scheduling can be used along with changes in the endhost TCP stack to minimize average flow completion times. We use RC3 [31] as our comparison-point for this objective (as it has better performance than RCP [21] and is simple to implement). In RC3 the senders aggressively send additional packets to quickly use up the available network capacity, but these packets are sent at lower priority levels to ensure that the regular traffic is not penalized. Therefore, it allows near-optimal bandwidth utilization, while maintaining the cautiousness of TCP.



Expt. Setup	Avg FCT (s)		
	TCP-FIFO	RC3-priorities	RC3-LSTF
I2 1Gbps-10Gbps at 30% util.	0.145	0.083	0.082
I2 1Gbps-10Gbps at 50% util.	0.159	0.094	0.089
I2 1Gbps-10Gbps at 70% util.	0.180	0.107	0.102
I2 1Gbps-1Gbps at 30% util.	0.134	0.075	0.073
I2 / 10 at 30% util.	0.32	0.215	0.233
Rocketfuel at 30% util.	0.171	0.102	0.101

Figure 10: The graph shows the mean FCT bucketed by flow size for the I2 1Gbps-10Gbps topology with 30% utilization for regular TCP using FIFO and for RC3 using priorities and LSTF. The legend indicates the mean FCT across all flows. The table indicates the mean FCTs for varying settings.

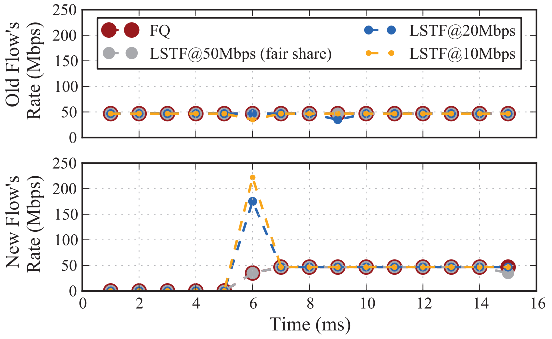


Figure 11: 20 flows share a single bottleneck link of 1Gbps and a 21st flow is added after 5ms. The graph shows the rate allocations for an old flow and the new flow with Fair Queuing and for LSTF with varying r_{est}.

Slack Initialization: The slack for a packet p is initialized as $slack(p) = prio_{rc3} * D$, where $prio_{rc3}$ is the priority of the packet assigned by RC3 and D is a value much larger than the queuing delay seen by any packet in the network. We use a value of $D = 1$ sec for our simulations.

Evaluation: To evaluate RC3 with LSTF, we reuse the ns-3 [7] implementation of RC3 (along with the same TCP parameters used by RC3, such as an initial congestion window of 4), and implement LSTF in ns-3. Figure 10 shows our results. We see that using LSTF with RC3 performs comparable to (and often slightly better than) using priorities with RC3, both giving significantly lower FCTs than regular TCP with FIFO.

C Fairness Deep Dive

C.1 Understanding how LSTF provides long-term fairness

The reason behind why any slack assignment with $r_{est} < r^*$ leads to convergence to fairness is quite straight-forward and is explained by the control experiment shown in

Figure 11. 20 long-lived TCP flows share a single bottleneck link of 1Gbps (giving a fair share rate of 50Mbps) and a 21st flow is added after 5ms. Since the first 20 flows have started early, the queue at the bottleneck link already contains packets belonging to these flows.

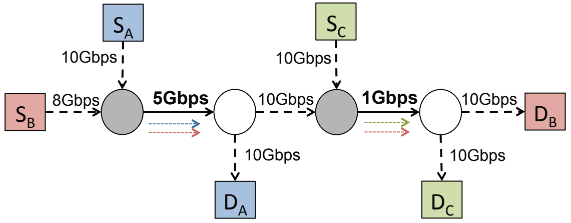
When $r_{est} = 50Mbps$, the actual queuing delay experienced by a packet is almost equal to the slack value assigned to it. Therefore, at any given point of time, the first packet of each flow present in the queue will have a slack value which is approximately equal to zero. The next packet of each flow will have a higher slack value (around 1500bytes/50Mbps = 0.24ms). By the time the corresponding first packets of every flow in the queue have been transmitted, the slack values of the next packet would also have been reduced to zero and so on. It therefore produces a round-robin pattern for scheduling packets across flows, as is done by FQ. Therefore, when the 21st flow starts at 5ms, with the first packet coming in with zero slack, the next one with 0.24ms slack and so on, it immediately starts following the round-robin pattern as well.

However, when r_{est} is smaller than $50Mbps$, then the packets of the old flows already present in the queue have a higher slack value than what they actually experience in the network. The first packet of every flow in the queue therefore has a slack which is more than 0 when the 21st flow comes in at 5ms. The earlier packets of the new flow therefore get precedence over any of the existing packets of the old flows, resulting in the spike in the rate allocated to the new flow as shown in Figure 11. Nonetheless, with the slack of every newly arriving packet of the 21st flow being higher than the previous one and with the slack of the already queued up packet decreasing with time, the slack value of the first packet in the queue for new flow and the old flows soon *catch up* with each other and the schedule starts following a round robin pattern again. The closer r_{est} is to the fair-share rate, the sooner the slack values of the old flows and the new flow *catch up* with each other. The duration for which a packet ends up waiting in the queue is upper-bounded by the time it would have waited, had all the flows arrived at the same time and were being serviced at their fair share rate.

C.2 Weighted Fairness with multiple-bottlenecks

One can see how the above logic can be extended for achieving weighted fairness. Moreover, when a packet sees multiple bottlenecks, the slack update (subtraction of the duration for which the packet waits) at the first bottleneck ensures that the next bottleneck takes into account the rate-limiting happening at the first one and the packets are given precedence accordingly.

We did a control experiment to evaluate weighted fairness with LSTF on a multi-bottleneck topology. We started three UDP flows with a start-time jitter between 0 and 1ms, on the topology as shown in Figure 12. We ran the simulation for 30ms and computed the throughput each flow received for the last 15ms. We varied the values of r_{est} used for assigning slacks to each flow, relative to one another, to assign different weights to different flows. For example, r_{est} assignment $\{A:900Mbps, B:100Mbps, C:100Mbps\}$ results in Flow A getting 9 times more share on the 5Gbps link than Flow B, with Flows B and C sharing the 1Gbps link equally. We compute the expected throughput based on the assigned r_{est} values and find that the throughput actually achieved is almost the same, as shown in the table.



r_{est} value (Mbps)			Expected Throughput (Mbps)			Achieved Throughput (Mbps)		
A	B	C	A	B	C	A	B	C
2000	100	100	4761	238	762	4762	238	763
900	100	100	4500	500	500	4499	501	500
500	100	100	4167	500	500	4166	501	500
200	100	100	3333	500	500	3333	501	500
100	100	100	2500	500	500	2500	500	501
100	100	500	2500	167	833	2500	167	834

Figure 12: Weighted Fairness on a multi-bottleneck topology (drawn above). The link capacities and the source/destination of each flow are indicated in the figure. Flows A and B share a 5Gbps link and then Flows B and C share a 1Gbps link.

Refresh Threshold (ms)	**Avg FCT across bytes (s)**	**Avg RTT across bytes (s)**
10	3.578	0.143
20	3.739	0.139
30	3.954	0.135
40	4.079	0.132

Table 4: Effect of varying refresh threshold on I2/10 topology at 70% utilization running LSTF (r_{est} = 10Mbps) with Edge-CoDel.

D Effect of Refresh Threshold on Edge-CoDel

To see whether our results for Edge-CoDel were highly dependent on the refresh threshold value, consider Table 4 which shows the average FCT and RTT values for varying refresh thresholds. We find that there are very minor differences in the results as we vary this threshold, because the dominating cause for refreshing the interval is when a packet sees a queuing delay less than the CoDel target. However, the general trend is that increasing the refresh threshold increases the FCT and decreases the RTT. This is because with increasing refresh threshold, the interval is reset to the larger 100ms value less frequently. This results in more packet drops for the long flows, causing an increase in FCTs, but a decrease in the RTT values.

Maglev: A Fast and Reliable Software Network Load Balancer

Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith,
Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney,
Wentao Shang[†*] and Jinnah Dylan Hosein[‡*]

Google Inc. [†]UCLA [‡]SpaceX
maglev-nsdi@google.com

Abstract

Maglev is Google's network load balancer. It is a large distributed software system that runs on commodity Linux servers. Unlike traditional hardware network load balancers, it does not require a specialized physical rack deployment, and its capacity can be easily adjusted by adding or removing servers. Network routers distribute packets evenly to the Maglev machines via Equal Cost Multipath (ECMP); each Maglev machine then matches the packets to their corresponding services and spreads them evenly to the service endpoints. To accommodate high and ever-increasing traffic, Maglev is specifically optimized for packet processing performance. A single Maglev machine is able to saturate a 10Gbps link with small packets. Maglev is also equipped with consistent hashing and connection tracking features, to minimize the negative impact of unexpected faults and failures on connection-oriented protocols. Maglev has been serving Google's traffic since 2008. It has sustained the rapid global growth of Google services, and it also provides network load balancing for Google Cloud Platform.

1 Introduction

Google is a major source of global Internet traffic [29, 30]. It provides hundreds of user-facing services, in addition to many more services hosted on the rapidly growing *Cloud Platform* [6]. Popular Google services such as *Google Search* and *Gmail* receive millions of queries per second from around the globe, putting tremendous demand on the underlying serving infrastructure.

To meet such high demand at low latency, a Google service is hosted on a number of servers located in multiple clusters around the world. Within each cluster, it is essential to distribute traffic load evenly across these servers in order to utilize resources efficiently so that no single server gets overloaded. As a result, network load balancers form a critical component of Google's production network infrastructure.

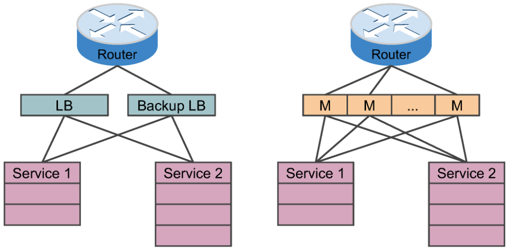


Figure 1: Hardware load balancer and Maglev.

A network load balancer is typically composed of multiple devices logically located between routers and service endpoints (generally TCP or UDP servers), as shown in Figure 1. The load balancer is responsible for matching each packet to its corresponding service and forwarding it to one of that service's endpoints.

Network load balancers have traditionally been implemented as dedicated hardware devices [1, 2, 3, 5, 9, 12, 13], an approach that has several limitations. First, their scalability is generally constrained by the maximum capacity of a single unit, making it impossible to keep up with Google's traffic growth. Second, they do not meet Google's requirements for high availability. Though often deployed in pairs to avoid single points of failure, they only provide *1+1 redundancy*. Third, they lack the flexibility and programmability needed for quick iteration, as it is usually difficult, if not impossible, to modify a hardware load balancer. Fourth, they are costly to upgrade. Augmenting the capacity of a hardware load balancer usually involves purchasing new hardware as well as physically deploying it. Because of all these limitations, we investigated and pursued alternative solutions.

With all services hosted in clusters full of commodity servers, we can instead build the network load balancer as a distributed software system running on these servers. A software load balancing system has many advantages

[*] Work was done while at Google.

over its hardware counterpart. We can address scalability by adopting the *scale-out* model, where the capacity of the load balancer can be improved by increasing the number of machines in the system: through ECMP forwarding, traffic can be evenly distributed across all machines. Availability and reliability are enhanced as the system provides *N+1 redundancy*. By controlling the entire system ourselves, we can quickly add, test, and deploy new features. Meanwhile, deployment of the load balancers themselves is greatly simplified: the system uses only existing servers inside the clusters. We can also divide services between multiple *shards* of load balancers in the same cluster in order to achieve performance isolation.

Despite all the benefits, the design and implementation of a software network load balancer are highly complex and challenging. First, each individual machine in the system must provide high throughput. Let N be the number of machines in the system and T be the maximum throughput of a single machine. The maximum capacity of the system is bounded by $N \times T$. If T is not high enough, it will be uneconomical for the system to provide enough capacity for all services [22]. The system as a whole must also provide *connection persistence*: packets belonging to the same connection should always be directed to the same service endpoint. This ensures quality of service as clusters are very dynamic and failures are quite common [23, 40].

This paper presents Maglev, a fast and reliable software network load balancing system. Maglev has been a critical component of Google's frontend serving infrastructure since 2008, and currently serves almost all of Google's incoming user traffic. By exploiting recent advances in high-speed server networking techniques [18, 41, 35, 31], each Maglev machine is able to achieve line-rate throughput with small packets. Through *consistent hashing* and *connection tracking*, Maglev provides reliable packet delivery despite frequent changes and unexpected failures. While some of the techniques described in this paper have existed for years, this paper shows how to build an operational system using these techniques. The major contributions of this paper are to: 1) present the design and implementation of Maglev, 2) share experiences of operating Maglev at a global scale, and 3) demonstrate the capability of Maglev through extensive evaluations.

2 System Overview

This section provides an overview of how Maglev works as a network load balancer. We give a brief introduction to Google's frontend serving architecture, followed by a description of how the Maglev system is configured.

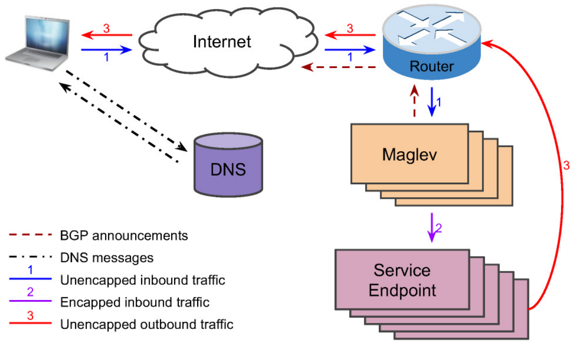

3
1
Internet
3
1
Router
1
DNS
Maglev
3
2
Service Endpoint
BGP announcements
DNS messages
1 Unencapped inbound traffic
2 Encapped inbound traffic
3 Unencapped outbound traffic

Figure 2: Maglev packet flow.

2.1 Frontend Serving Architecture

Maglev is deployed in Google's frontend-serving locations, including clusters of varying sizes. For simplicity, we only focus on the setup in the smaller clusters in this paper, and briefly describe the larger cluster setup below. Figure 2 shows an overview of Google's frontend serving architecture in the small cluster setup.

Every Google service has one or more *Virtual IP addresses* (VIPs). A VIP is different from a physical IP in that it is not assigned to a specific network interface, but rather served by multiple service endpoints behind Maglev. Maglev associates each VIP with a set of service endpoints and announces it to the router over BGP; the router in turn announces the VIP to Google's backbone. Aggregations of the VIP networks are announced to the Internet to make them globally accessible. Maglev handles both IPv4 and IPv6 traffic, and all the discussion below applies equally to both.

When a user tries to access a Google service served on *www.google.com*, her browser first issues a DNS query, which gets a response (possibly cached) from one of Google's authoritative DNS servers. The DNS server assigns the user to a nearby frontend location taking into account both her geolocation and the current load at each location, and returns a VIP belonging to the selected location in response [16]. The browser will then try to establish a new connection with the VIP.

When the router receives a VIP packet, it forwards the packet to one of the Maglev machines in the cluster through ECMP, since all Maglev machines announce the VIP with the same cost. When the Maglev machine receives the packet, it selects an endpoint from the set of service endpoints associated with the VIP, and encapsulates the packet using *Generic Routing Encapsulation* (GRE) with the outer IP header destined to the endpoint.

When the packet arrives at the selected service endpoint, it is decapsulated and consumed. The response, when ready, is put into an IP packet with the source address being the VIP and the destination address being

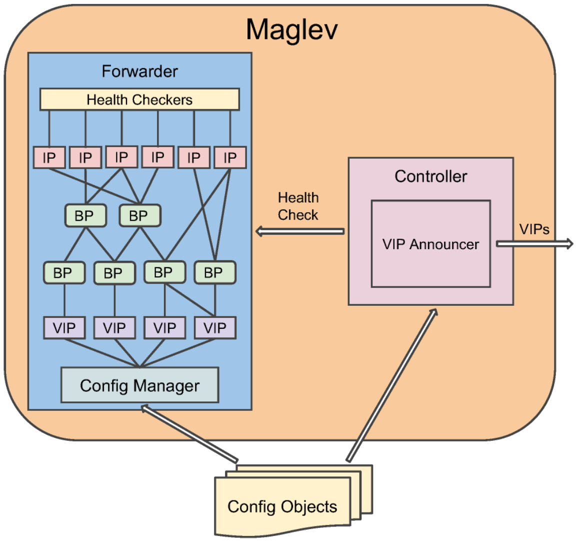


Figure 3: Maglev config (BP stands for backend pool).

the IP of the user. We use *Direct Server Return* (DSR) to send responses directly to the router so that Maglev does not need to handle returning packets, which are typically larger in size. This paper focuses on the load balancing of incoming user traffic. The implementation of DSR is out of the scope of this paper.

The setup for large clusters is more complicated: to build clusters at scale, we want to avoid the need to place Maglev machines in the same layer-2 domain as the router, so hardware encapsulators are deployed behind the router, which tunnel packets from routers to Maglev machines.

2.2 Maglev Configuration

As described in the previous subsection, Maglev is responsible for announcing VIPs to the router and forwarding VIP traffic to the service endpoints. Therefore, each Maglev machine contains a controller and a forwarder as depicted in Figure 3. Both the controller and the forwarder learn the VIPs to be served from configuration objects, which are either read from files or received from external systems through RPC.

On each Maglev machine, the controller periodically checks the health status of the forwarder. Depending on the results, the controller decides whether to announce or withdraw all the VIPs via BGP. This ensures the router only forwards packets to healthy Maglev machines.

All VIP packets received by a Maglev machine are handled by the forwarder. At the forwarder, each VIP is configured with one or more backend pools. Unless otherwise specified, the backends for Maglev are service endpoints. A backend pool may contain the physical IP addresses of the service endpoints; it may also recursively contain other backend pools, so that a frequently-used set of backends does not need to be specified repeatedly. Each backend pool, depending on its specific requirements, is associated with one or more health checking methods with which all its backends are verified; packets will only be forwarded to the healthy backends. As the same server may be included in more than one backend pool, health checks are deduplicated by IP addresses to avoid extra overhead.

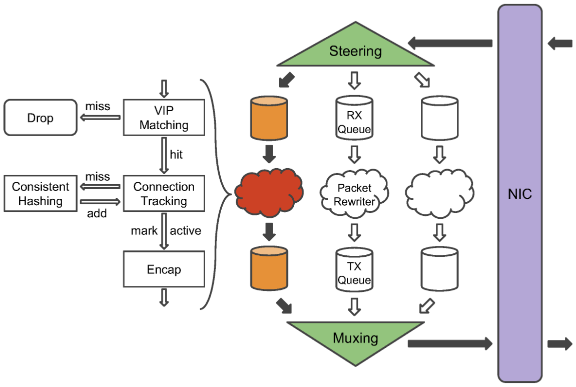


Figure 4: Maglev forwarder structure.

The forwarder's *config manager* is responsible for parsing and validating config objects before altering the forwarding behavior. All config updates are committed atomically. Configuration of Maglev machines within the same cluster may become temporarily out of sync due to delays in config push or health checks. However, consistent hashing will make connection flaps between Maglevs with similar backend pools mostly succeed even during these very short windows.

It is possible to deploy multiple *shards* of Maglevs in the same cluster. Different Maglev shards are configured differently and serve different sets of VIPs. Sharding is useful for providing performance isolation and ensuring quality of service. It is also good for testing new features without interfering with regular traffic. For simplicity, we assume one shard per cluster in this paper.

3 Forwarder Design and Implementation

The forwarder is a critical component of Maglev, as it needs to handle a huge number of packets quickly and reliably. This section explains the design and implementation details of the key modules of the Maglev forwarder, as well as the rationale behind the design.

3.1 Overall Structure

Figure 4 illustrates the overall structure of the Maglev forwarder. The forwarder receives packets from the

NIC (Network Interface Card), rewrites them with proper GRE/IP headers and then sends them back to the NIC. The Linux kernel is not involved in this process.

Packets received by the NIC are first processed by the *steering module* of the forwarder, which calculates the *5-tuple hash*[1] of the packets and assigns them to different *receiving queues* depending on the hash value. Each receiving queue is attached to a packet rewriter thread. The packet thread first tries to match each packet to a configured VIP. This step filters out unwanted packets not targeting any VIP. Then it recomputes the 5-tuple hash of the packet and looks up the hash value in the *connection tracking table* (covered in Section 3.3). We do not reuse the hash value from the steering module to avoid cross-thread synchronization.

The connection table stores backend selection results for recent connections. If a match is found and the selected backend is still healthy, the result is simply reused. Otherwise the thread consults the *consistent hashing module* (covered in Section 3.4) and selects a new backend for the packet; it also adds an entry to the connection table for future packets with the same 5-tuple. A packet is dropped if no backend is available. The forwarder maintains one connection table per packet thread to avoid access contention. After a backend is selected, the packet thread encapsulates the packet with proper GRE/IP headers and sends it to the attached *transmission queue*. The *muxing module* then polls all transmission queues and passes the packets to the NIC.

The steering module performs 5-tuple hashing instead of round-robin scheduling for two reasons. First, it helps lower the probability of packet reordering within a connection caused by varying processing speed of different packet threads. Second, with connection tracking, the forwarder only needs to perform backend selection once for each connection, saving clock cycles and eliminating the possibility of differing backend selection results caused by race conditions with backend health updates. In the rare cases where a given receiving queue fills up, the steering module falls back to round-robin scheduling and spreads packets to other available queues. This fallback mechanism is especially effective at handling large floods of packets with the same 5-tuple.

3.2 Fast Packet Processing

The Maglev forwarder needs to process packets as fast as possible in order to cost-effectively scale the serving capacity to the demands of Google's traffic. We engineered it to forward packets at line rate – typically 10Gbps in Google's clusters today. This translates to 813Kpps (packets per second) for 1500-byte IP packets.

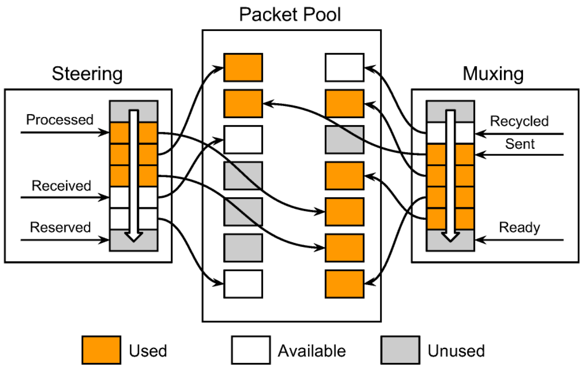


Figure 5: Packet movement into and out of the forwarder.

However, our requirements are much more stringent: we must handle very small packets effectively because incoming requests are typically small in size. Assuming IP packet size is 100 bytes on average, the forwarder must be able to process packets at 9.06Mpps. This subsection describes the key techniques we employed to reach and exceed this packet processing speed.

Maglev is a userspace application running on commodity Linux servers. Since the Linux kernel network stack is rather computationally expensive, and Maglev doesn't require any of the Linux stack's features, it is desirable to make Maglev bypass the kernel entirely for packet processing. With proper support from the NIC hardware, we have developed a mechanism to move packets between the forwarder and the NIC without any involvement of the kernel, as shown in Figure 5. When Maglev is started, it pre-allocates a packet pool that is shared between the NIC and the forwarder. Both the steering and muxing modules maintain a *ring queue* of pointers pointing to packets in the packet pool.

Both the steering and muxing modules maintain three pointers to the rings. At the receiving side, the NIC places newly received packets at the *received* pointer and advances it. The steering module distributes the received packets to packet threads and advances the *processed* pointer. It also reserves unused packets from the packet pool, places them into the ring and advances the *reserved* pointer. The three pointers chase one another as shown by the arrows. Similarly, on the sending side the NIC sends packets pointed to by the *sent* pointer and advances it. The muxing module places packets rewritten by packet threads into the ring and advances the *ready* pointer. It also returns packets already sent by the NIC back to the packet pool and advances the *recycled* pointer. Note that the packets are not copied anywhere by the forwarder.

To reduce the number of expensive boundary-crossing operations, we process packets in batches whenever possible. In addition, the packet threads do not share any

[1] The 5-tuple of a packet refers to the source IP, source port, destination IP, destination port and IP protocol number.

data with each other, preventing contention between them. We pin each packet thread to a dedicated CPU core to ensure best performance. With all these optimizations, Maglev is able to achieve line rate with small packets, as shown in Section 5.2.

Further, the latency that Maglev adds to the path taken by each packet is small. Normally it takes the packet thread about 350ns to process each packet on our standard servers. There are two special cases in which packet processing may take longer. Since the forwarder processes packets in batches, each batch is processed when it grows large enough or when a periodic timer expires. In practice we set the timer to be 50μs. Therefore if Maglev is significantly underloaded, a 50μs delay will be added to each packet in the worst case. One possible optimization to this case is to adjust batch sizes dynamically [32]. The other case where Maglev may add extra processing delay is when Maglev is overloaded. The maximum number of packets that Maglev can buffer is the size of the packet pool; beyond that the packets will be dropped by the NIC. Assuming the packet pool size is 3000 and the forwarder can process 10Mpps, it takes about 300μs to process all buffered packets. Hence a maximum of 300μs delay may be added to each packet if Maglev is heavily overloaded. Fortunately, this case can be avoided by proper capacity planning and adding Maglev machines as needed.

3.3 Backend Selection

Once a packet is matched to a VIP, we need to choose a backend for the packet from the VIP's backend pool. For connection-oriented protocols such as TCP, it is critical to send all packets of a connection to the same backend. We accomplish this with a two part strategy. First, we select a backend using a new form of consistent hashing which distributes traffic very evenly. Then we record the selection in a local connection tracking table.

Maglev's connection tracking table uses a fixed-size hash table mapping 5-tuple hash values of packets to backends. If the hash value of a packet does not exist in the table, Maglev will assign a backend to the packet and store the assignment in the table. Otherwise Maglev will simply reuse the previously assigned backend. This guarantees that packets belonging to the same connection are always sent to the same backend, as long as the backend is still able to serve them. Connection tracking comes in handy when the set of backends changes: for instance, when backends go up and down, are added or removed, or when the backend weights change.

However, per-Maglev connection tracking alone is insufficient in our distributed environment. First, it assumes all packets with the same 5-tuple are always sent to the same Maglev machine. Because the router in front of Maglev does not usually provide connection affinity, this assumption does not hold when the set of Maglev machines changes. Unfortunately, such changes are inevitable and may happen for various reasons. For example, when upgrading Maglevs in a cluster we do a rolling restart of machines, draining traffic from each one a few moments beforehand and restoring it once the Maglev starts serving again. This process may last over an hour, during which the set of Maglevs keeps changing. We also sometimes add, remove, or replace Maglev machines. All of these operations make standard ECMP implementations shuffle traffic on a large scale, leading to connections switching to different Maglevs in mid-stream. The new Maglevs will not have the correct connection table entries, so if backend changes occur at the same time, connections will break.

A second theoretical limitation is that the connection tracking table has finite space. The table may fill up under heavy load or SYN flood attacks. Since Maglev only evicts entries from the connection table when they are expired, once the table becomes full, we will need to select a backend for each packet that doesn't fit in the table. While in practice there is plenty of memory on a modern machine, in deployments where we share machines between Maglev and other services, we may need to sharply limit the connection table size.

If any of the above cases occur, we can no longer rely on connection tracking to handle backend changes. Thus Maglev also provides consistent hashing to ensure reliable packet delivery under such circumstances.

3.4 Consistent Hashing

One possible approach to address the limitations of connection tracking is to share connection state among all Maglev machines, for example in a distributed hash table as suggested in [34]. However, this would negatively affect forwarding performance – recall that connection states are not even shared among packet threads on the same Maglev machine to avoid contention.

A better-performing solution is to use local consistent hashing. The concept of consistent hashing [28] or rendezvous hashing [38] was first introduced in the 1990s. The idea is to generate a large lookup table with each backend taking a number of entries in the table. These methods provide two desirable properties that Maglev also needs for resilient backend selection:

- *load balancing*: each backend will receive an almost equal number of connections.

- *minimal disruption*: when the set of backends changes, a connection will likely be sent to the same backend as it was before.

Pseudocode 1 Populate Maglev hashing lookup table.

```
function POPULATE
    for each $i < N$ do $next[i] \leftarrow 0$ end for
    for each $j < M$ do $entry[j] \leftarrow -1$ end for
    $n \leftarrow 0$
    while true do
        for each $i < N$ do
            $c \leftarrow permutation[i][next[i]]$
            while $entry[c] \geq 0$ do
                $next[i] \leftarrow next[i] + 1$
                $c \leftarrow permutation[i][next[i]]$
            end while
            $entry[c] \leftarrow i$
            $next[i] \leftarrow next[i] + 1$
            $n \leftarrow n + 1$
            if $n = M$ then return end if
        end for
    end while
end function
```

Table 1: A sample consistent hash lookup table.

$B1$	$B2$	$B3$
2	1	5
4	5	6
6	2	7
1	6	1
3	3	2
5	7	3
7	4	4

Permutation tables for the backends.

Before	After
$B2$	$B1$
$B1$	$B1$
$B2$	$B1$
$B1$	$B1$
$B3$	$B3$
$B3$	$B3$
$B1$	$B3$

Lookup table before and after $B2$ is removed.

Both [28] and [38] prioritize minimal disruption over load balancing, as they were designed to optimize web caching on a small number of servers. However, Maglev takes the opposite approach for two reasons. First, it is critical for Maglev to balance load as evenly as possible among the backends. Otherwise the backends must be aggressively overprovisioned in order to accommodate the peak traffic. Maglev may have hundreds of backends for certain VIPs, our experiments show that both [28] and [38] will require a prohibitively large lookup table for each VIP to provide the level of load balancing that Maglev desires. Second, while minimizing lookup table disruptions is important, a small number of disruptions is tolerable by Maglev. Steady state, changes to the lookup table do not lead to connection resets because connections' affinity to Maglev machines does not change at the same time. When connections' affinity to Maglevs does change, resets are proportional to the number of lookup table disruptions.

With these considerations in mind, we developed a new consistent hashing algorithm, which we call *Maglev hashing*. The basic idea of Maglev hashing is to assign a preference list of all the lookup table positions to each backend. Then all the backends take turns filling their most-preferred table positions that are still empty, until the lookup table is completely filled in. Hence, Maglev hashing gives an almost equal share of the lookup table to each of the backends. Heterogeneous backend weights can be achieved by altering the relative frequency of the backends' turns; the implementation details are not described in this paper.

Let M be the size of the lookup table. The preference list for backend i is stored in $permutation[i]$, which is a random permutation of array $(0..M-1)$. As an efficient way of generating $permutation[i]$, each backend is assigned a unique name. We first hash the backend name using two different hashing functions to generate two numbers *offset* and *skip*. Then we generate $permutation[i]$ using these numbers as follows:

$$\begin{aligned} \mathit{offset} &\leftarrow h_1(name[i]) \bmod M \\ \mathit{skip} &\leftarrow h_2(name[i]) \bmod (M-1)+1 \\ permutation[i][j] &\leftarrow (\mathit{offset} + j \times \mathit{skip}) \bmod M \end{aligned}$$

M must be a prime number so that all values of *skip* are relatively prime to it. Let N be the size of a VIP's backend pool. Its lookup table is populated using Pseudocode 1. We use $next[i]$ to track the next index in the permutation to be considered for backend i; the final lookup table is stored in the array $entry$. In the body of the outer *while* loop, we iterate through all the backends. For each backend i we find a candidate index c from $permutation[i]$ which has not been filled yet, and fill it with the backend. The loop keeps going until all entries in the table have been filled.

The algorithm is guaranteed to finish. Its worst case time complexity is $O(M^2)$ which only happens if there are as many backends as lookup table entries and all the backends hash to the same permutation. To avoid this happening we always choose M such that $M \gg N$. The average time complexity is $O(M \log M)$ because at step n we expect the algorithm to take $\frac{M}{M-n}$ tries to find an empty candidate index, so the total number of steps is $\sum_{n=1}^{M} \frac{M}{n}$. Each backend will take either $\lfloor \frac{M}{N} \rfloor$ or $\lceil \frac{M}{N} \rceil$ entries in the lookup table. Therefore the number of entries occupied by different backends will differ by at most 1. In practice, we choose M to be larger than $100 \times N$ to ensure at most a 1% difference in hash space assigned to backends. Other methods of generating random permutations, such as the Fisher-Yates Shuffle [20], generate better quality permutations using more state, and would work fine here as well.

We use the example in Table 1 to illustrate how Maglev hashing works. Assume there are 3 backends, the lookup table size is 7, and the (offset, skip) pairs of the three backends are (3, 4), (0, 2) and (3, 1). The generated permutation tables are shown in the left column, and the lookup tables before and after backend $B2$ is removed are presented in the right column. As the example shows, the lookup tables are evenly balanced among the backends both with and without $B2$. After $B2$ is removed, aside from updating all of the entries that contained $B2$, only one other entry (row 7) needs to be changed. In practice, with larger lookup tables, Maglev hashing is fairly resilient to backend changes, as we show in Section 5.3.

4 Operational Experience

Maglev is a highly complex distributed system that has been serving Google for over six years. We have learned a lot while operating it at a global scale. This section describes how Maglev has evolved over the years to accommodate our changing requirements, and some of the tools we've built to monitor and debug the system.

4.1 Evolution of Maglev

Today's Maglev differs in many details from the original system. Most of the changes, such as the addition of IPv6 support, happened smoothly as a result of the extensible software architecture. This subsection discusses two major changes to the implementation and deployment of Maglev since its birth.

4.1.1 Failover

Maglev machines were originally deployed in active-passive pairs to provide failure resilience, as were the hardware load balancers they replaced. Only active machines served traffic in normal situations. When an active machine became unhealthy, its passive counterpart would take over and start serving. Connections were usually uninterrupted during this process thanks to Maglev hashing, but there were some drawbacks to this setup. It used resources inefficiently, since half of the machines sat idle at all times. It also prevented us from scaling any VIP past the capacity of a single Maglev machine. Finally, coordination between active and passive machines was complex. In this setup, the machines' announcers would monitor each other's health and serving priority, escalating their own BGP priority if they lost sight of each other, with various tie-breaking mechanisms.

We gained a great deal of capacity, efficiency, and operational simplicity by moving to an ECMP model. While Maglev hashing continues to protect us against occasional ECMP flaps, we can multiply the capacity of a VIP by the maximum ECMP set size of the routers, and all machines can be fully utilized.

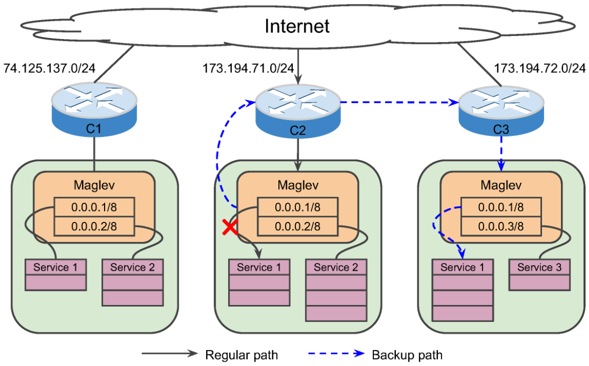


Figure 6: Maglev VIP matching.

4.1.2 Packet Processing

Maglev originally used the Linux kernel network stack for packet processing. It had to interact with the NIC using kernel sockets, which brought significant overhead to packet processing including hardware and software interrupts, context switches and system calls [26]. Each packet also had to be copied from kernel to userspace and back again, which incurred additional overhead. Maglev does not require a TCP/IP stack, but only needs to find a proper backend for each packet and encapsulate it using GRE. Therefore we lost no functionality and greatly improved performance when we introduced the kernel bypass mechanism – the throughput of each Maglev machine is improved by more than a factor of five.

4.2 VIP Matching

In Google's production networks, each cluster is assigned an external IP prefix that is globally routable. For example, cluster C1 in Figure 6 has prefix 74.125.137.0/24. The same service is configured as different VIPs in different clusters, and the user is directed to one of them by DNS. For instance, Service1 is configured as 74.125.137.1 in C1 and 173.194.71.1 in C2.

Google has several different classes of clusters, serving different sets of VIPs. External prefix lengths are the same for clusters of the same class, but may be different for different cluster classes. Sometimes, in emergencies, we need to redirect traffic to a different cluster via Maglev encapsulation. Therefore, we need the target Maglevs to be able to correctly classify traffic for arbitrary other clusters. One possible solution is to define all VIPs in all the clusters that may receive redirected traffic, but that would cause synchronization and scalability issues.

Instead, we implemented a special numbering rule and a novel VIP matching mechanism to cope with the problem. For each cluster class, we assign each VIP the same *suffix* across all clusters of that class. Then we use a prefix/suffix matching mechanism for VIP matching. First, the incoming packet goes through longest prefix matching, to determine which cluster class it was destined for. Then it goes through longest suffix matching specific to that cluster class, to determine which backend pool it should be sent to. In order to reduce the need to keep configs globally in sync on a tight time scale, we preconfigure maglevs with a large prefix group for each cluster class, from which prefixes for new clusters of the same class are allocated. This way a Maglev can correctly serve traffic originally destined for a cluster that it has never heard of.

As a result, each VIP is configured as a <Prefix Group, IP suffix, port, protocol> tuple. Take Figure 6 as an example. Assuming C2 and C3 are of the same class, if a packet towards 173.194.71.1 is received in C2 but Maglev determines none of the endpoints in C2 can serve the packet, it will encapsulate and tunnel the packet towards the VIP address in C3 for the same service (173.194.72.1). Then a Maglev in C3 will decapsulate the packet and match the inner packet to Service1 using prefix/suffix matching, and the packet will be served by an endpoint in C3 instead.

This VIP matching mechanism is specific to Google's production setup, but it provides a good example of the value of rapid prototyping and iteration that a software-based load balancer can offer.

4.3 Fragment Handling

One special case that is not covered by the system described so far is IP fragmentation. Fragments require special treatment because Maglev performs 5-tuple hashing for most VIPs, but fragments do not all contain the full 5-tuple. For example, if a large datagram is split into two fragments, the first fragment will contain both L3 and L4 headers while the second will only contain the L3 header. Thus when Maglev receives a non-first fragment, it cannot make the correct forwarding decision based only on that packet's headers.

Maglev must satisfy two requirements in order to handle fragments correctly. First, all fragments of the same datagram must be received by the same Maglev. Second, the Maglev must make consistent backend selection decisions for unfragmented packets, first fragments, and non-first fragments.

In general, we cannot rely on the hardware in front of Maglev to satisfy the first requirement on its own. For example, some routers use 5-tuple hashing for first fragments and 3-tuple for non-first fragments. We therefore implemented a generic solution in Maglev to cope with any fragment hashing behavior. Each Maglev is configured with a special backend pool consisting of all Maglevs within the cluster. Upon receipt of a fragment, Maglev computes its 3-tuple hash using the L3 header and forwards it to a Maglev from the pool based on the hash value. Since all fragments belonging to the same datagram contain the same 3-tuple, they are guaranteed to be redirected to the same Maglev. We use the GRE recursion control field to ensure that fragments are only redirected once.

To meet the second requirement, Maglev uses the same backend selection algorithm to choose a backend for unfragmented packets and second-hop first fragments (usually on different Maglev instances.) It maintains a fixed-size fragment table which records forwarding decisions for first fragments. When a second-hop non-first fragment is received by the same machine, Maglev looks it up in the fragment table and forwards it immediately if a match is found; otherwise it is cached in the fragment table until the first one is received or the entry expires.

This approach has two limitations: it introduces extra hops to fragmented packets, which can potentially lead to packet reordering. It also requires extra memory to buffer non-first fragments. Since packet reordering may happen anywhere in the network, we rely on the endpoints to handle out-of-order packets. In practice only a few VIPs are allowed to receive fragments, and we are easily able to provide a big enough fragment table to handle them.

4.4 Monitoring and Debugging

We consistently monitor the health and behavior of Maglev as we do any other production system – for example, we use both black box and white box monitoring. Our black box monitoring consists of agents all over the world which periodically check the reachability and latency of the configured VIPs. For our white box monitoring, we export various metrics from each Maglev machine via an HTTP server, and the monitoring system periodically queries each server to learn the latest Maglev serving status details. The system sends alerts when it observes abnormal behavior.

Due to Maglev's distributed nature, multiple paths exist from the router through Maglev to the service endpoints. However, debugging is much easier when we are able to discern the exact path that a specific packet takes through the network. Thus we developed the *packet-tracer* tool, similar to X-trace [21]. Packet-tracer constructs and sends specially marked Maglev-recognizable payloads with specified L3 and L4 headers. The payloads contain receiver IP addresses to which Maglev sends debugging information. The packets usually target a specific VIP and are routed normally to our frontend locations. When a Maglev machine receives a packet-tracer packet, it forwards the packet as usual, while also sending debugging information, including its machine name and the selected backend, to the specified receiver.

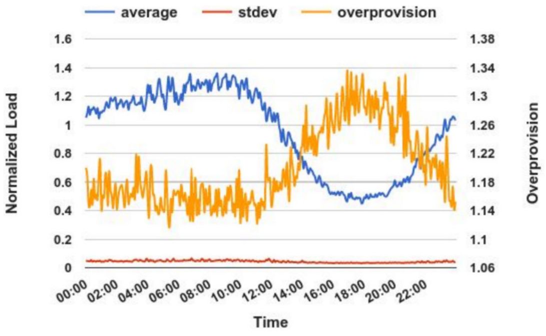


Figure 7: Average, standard deviation and coefficient of variation of normalized load on all service endpoints in one cluster on a typical day.

Packet-tracer packets are rate-limited by Maglev, as they are expensive to process. This tool is extremely helpful in debugging production issues, especially when there is more than one Maglev machine on the path, as happens in the case of fragment redirection.

5 Evaluation

In this section we evaluate Maglev's efficiency and performance. We present results from one of Google's production clusters, as well as some microbenchmarks.

5.1 Load Balancing

As a network load balancer, Maglev's major responsibility is to distribute traffic evenly across multiple service endpoints. To illustrate the load balancing performance of Maglev, we collected *connections per second* (*cps*) data from 458 endpoints in a cluster located in Europe. The data is aggregated from multiple HTTP services including Web Search. The granularity of data collection is 5 minutes, and the load is normalized by the average cps throughout the day. Figure 7 shows the average and standard deviation of the load across all endpoints on a typical day. The traffic load exhibits a clear diurnal pattern. The standard deviation is always small compared to the average load; the coefficient of variation is between 6% and 7% most of the time.

Figure 7 also presents the overprovision factor computed as the maximum load over the average load at each time point. It is an important metric because we must ensure even the busiest endpoints will always have enough capacity to serve all the traffic. The overprovision factor is less than 1.2 over 60% of the time. It is notably higher during off-peak hours, which is the expected behavior because it is harder to balance the load when there is less traffic. Besides, a higher overprovision factor during off-peak hours does not require the addition of Maglev machines. This provides a guideline of how much to overprovision at this specific location.

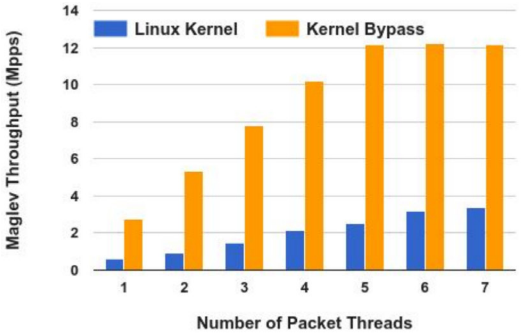


Figure 8: Throughput with and without kernel bypass.

5.2 Single Machine Throughput

Since each Maglev machine receives a roughly equal amount of traffic through ECMP, the overall throughput of Maglev can be estimated as the number of Maglev machines times the throughput of each single machine. The more traffic each machine can handle, the fewer machines will be required to provide the same frontend capacity. Thus single machine throughput is essential to the efficiency of the system.

The throughput of a Maglev machine is affected by many factors, including the number of packet threads, NIC speed, and traffic type. In this subsection we report results from a small testbed to evaluate the packet processing capability of a Maglev machine under various conditions. Unless otherwise specified, all experiments are conducted on servers equipped with two 8-core recent server-class CPUs, one 10Gbps NIC and 128GB of memory. We only use one CPU for Maglev. Everything else, including the operating system, runs on the other CPU. The testbed consists of two senders, two receivers and one Maglev machine located in the same Ethernet domain. The senders slowly increase their sending rates, and the throughput of Maglev is recorded as the maximum number of packets per second (pps)[2] that Maglev can handle before starting to drop packets. We use two senders to ensure Maglev eventually gets overloaded.

5.2.1 Kernel Bypass

In this experiment, we run Maglev in both vanilla Linux network stack mode as well as kernel bypass mode to evaluate the impact of kernel bypass on the throughput of

[2] Note that we report throughput by pps instead of bps because the effect of packet size on the pps throughput is negligible. Hence we measure the pps throughput using minimum-sized packets. The bps throughput is equal to $min(pps \times packet\_size, line\_rate\_bps)$.

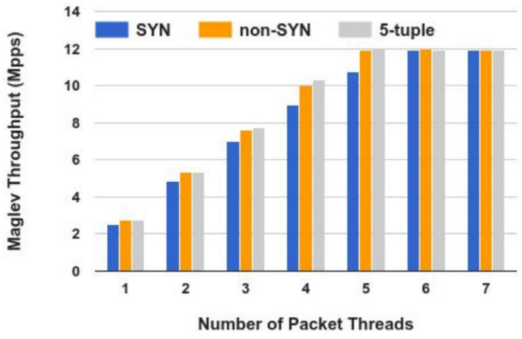


Figure 9: Throughput with different TCP packet types.

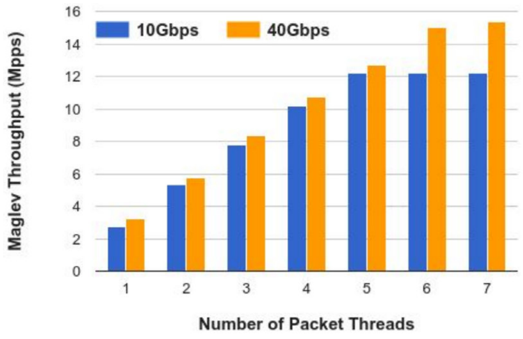


Figure 10: Throughput with different NIC speeds.

Maglev. The senders are configured to send minimum-sized UDP packets from different source ports so that they are not assigned to the same packet thread by the steering module. Due to limitations of the test environment, the minimum size of UDP packets the senders can send is 52 bytes, slightly larger than the theoretical minimum for Ethernet. We vary the number of packet threads in each run of the experiment. Each packet thread is pinned to a dedicated CPU core (as we do in production) to ensure best performance. We use one core for steering and muxing, thus there can be at most 7 packet threads. We measure Maglev's throughput with and without kernel bypass and present the results in Figure 8.

The figure shows the clear advantage of running Maglev in kernel bypass mode. There, Maglev is the bottleneck when there are no more than 4 packet threads; its throughput increases with the number of packet threads. When there are 5 or more packet threads, however, the NIC becomes the bottleneck. On the other hand, Maglev is always the bottleneck when using the vanilla Linux network stack, and the maximum throughput achieved is less than 30% that of kernel bypass.

5.2.2 Traffic Type

Depending on the code execution paths within a packet thread, Maglev handles different types of traffic at different speeds. For example, a packet thread needs to select a backend for a TCP SYN packet and record it in the connection tracking table; it only needs to do a lookup in the connection tracking table for non-SYN packets. In this experiment we measure how fast Maglev handles different types of TCP packets.

Three traffic types are considered: SYN, non-SYN and constant-5-tuple. For SYN and non-SYN experiments, only SYN and non-SYN TCP packets are sent, respectively. The SYN experiment shows how Maglev behaves during SYN flood attacks, while the non-SYN experiment shows how Maglev works with regular TCP traffic, performing backend selection once and using connection tracking afterwards. For the constant-5-tuple experiment, all packets contain the same L3 and L4 headers. This is a special case because the steering module generally tries to send packets with the same 5-tuple to the same packet thread, and only spreads them to other threads when the chosen one is full. The senders vary the source ports for SYN and non-SYN experiments to generate different 5-tuples, but always use the same source port for the constant-5-tuple experiment. They always send minimum-sized TCP packets, which are 64 bytes in our test environment.

As in the previous experiment, Maglev reaches the NIC's capacity with 5 packet threads in the non-SYN and constant-5-tuple experiments. However, for SYN packets, we see that Maglev needs 6 packet threads to saturate the NIC. This is because Maglev needs to perform backend selection for every SYN packet. Maglev performs best under constant-5-tuple traffic, showing that the steering module can effectively steer poorly-distributed packet patterns. Since all packets have the same 5-tuple, their connection tracking information always stays in the CPU cache, ensuring the highest throughput. For non-SYN packets, there are sporadic cache misses for connection tracking lookup, and so the throughput is slightly lower than that for constant-5-tuple traffic when there are fewer than 5 packet threads.

5.2.3 NIC Speed

In the previous experiments, the NIC is the bottleneck as it is saturated by 5 packet threads. To understand Maglev's full capability, this experiment evaluates its throughput using a faster NIC. Instead of the 10Gbps NIC, we install a 40Gbps NIC on the Maglev machine, and use the same setup as in Section 5.2.1. The results are illustrated in Figure 10. When there are no more than 5 packet threads, the 40Gbps NIC provides slightly higher throughput as its chip is faster than the 10Gbps one. However, the throughput growth for the 40Gbps NIC does not slow down until 7 packet threads are used.

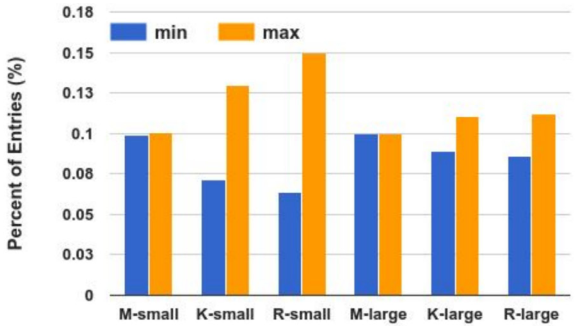


Figure 11: Load balancing efficiency of different hashing methods. M, K and R stand for Maglev, Karger and Rendezvous, respectively. Lookup table size is 65537 for *small* and 655373 for *large*.

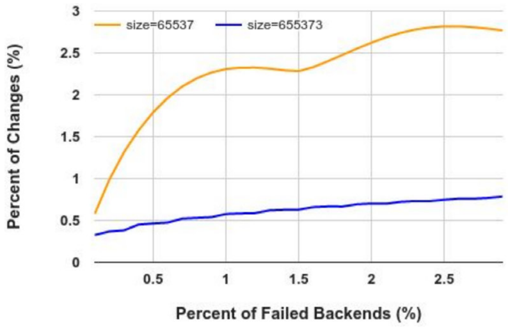


Figure 12: Resilience of Maglev hashing to backend changes.

Because the NIC is no longer the bottleneck, this figure shows the upper bound of Maglev throughput with the current hardware, which is slightly higher than 15Mpps. In fact, the bottleneck here is the Maglev steering module, which will be our focus of optimization when we switch to 40Gbps NICs in the future.

5.3 Consistent Hashing

In this experiment we evaluate Maglev hashing and compare it against *Karger* [28] and *Rendezvous* [38] hashing. We are interested in two metrics: load balancing efficiency and resilience to backend changes.

To evaluate the load balancing efficiency of the methods, we populate one lookup table using each method, and count the number of table entries assigned to each backend. We set the total number of backends to be 1000 and the lookup table size to be 65537 and 655373[3]. For Karger we set the number of views to be 1000. Figure 11 presents the maximum and minimum percent of entries per backend for each method and table size.

As expected, Maglev hashing provides almost perfect load balancing no matter what the table size is. When table size is 65537, Karger and Rendezvous require backends to be overprovisioned by 29.7% and 49.5% respectively to accommodate the imbalanced traffic. The numbers drop to 10.3% and 12.3% as the table size grows to 655373. Since there is one lookup table per VIP, the table size must be limited in order to scale the number of VIPs. Thus Karger and Rendezvous are not suitable for Maglev's load balancing needs.

Another important metric for consistent hashing is resilience to backend changes. Both Karger and Rendezvous guarantee that when some backends fail, the entries for the remaining backends will not be affected. Therefore we only evaluate this metric for Maglev. Figure 12 presents the percent of changed table entries as a function of the percent of concurrent backend failures. We set the number of backends to be 1000. For each failure number k, we randomly remove k backends from the pool, regenerate the lookup table and compute the percent of changed entries. We repeat the experiment 200 times for each k value and report the average results.

Figure 12 shows that the ratio of changed entries increases with the number of concurrent failures. Maglev hashing is more resilient to backend changes when the table size is larger. In practice we use 65537 as the default table size because we expect concurrent backend failures to be rare, and we still have connection tracking as the primary means of protection. In addition, microbenchmarks show that the lookup table generation time increases from 1.8ms to 22.9ms as the table size grows from 65537 to 655373, which prevents us from increasing the table size indefinitely.

6 Related Work

Unlike traditional hardware load balancers [1, 2, 3, 5, 9, 12, 13], Maglev is a distributed software system which runs on commodity servers. Hardware load balancers are usually deployed as active-passive pairs. Maglev provides better efficiency and resiliency by running all servers in active mode. In addition, upgrading hardware load balancer capacity requires purchasing new hardware as well as physically deploying it, making on demand capacity adjustment difficult. On the other hand, Maglev's capacity can easily be adjusted up or down without causing any service disruption. Some hardware vendors also provide load balancing software that runs in virtualized environments. Maglev provides much higher throughput than these virtual load balancers.

Ananta [34] is a distributed software load balancer. Like Maglev, it employs ECMP to scale out the system and uses a flow table to achieve connection affinity.

[3] There is no special significance to these numbers except that they need to be prime.

However, it does not provide a concrete mechanism to handle changes to the load balancer pool gracefully, and it is not specially optimized for single machine performance. Maglev does not have a component similar to Ananta's HostAgent which provides NAT services, but there is an external system (not described here) that offers similar functionality. Ananta allows most internal VIP traffic to bypass the load balancer. Maglev does not provide a similar feature because it has enough capacity for the internal traffic. Embrane [4] is a similar system developed for virtual environments. However, its throughput optimization can be difficult due to the limitations of virtualization. Duet [22] is a hybrid hardware and software load balancer which aims to address the low throughput issue of pure software load balancers. Maglev is able to achieve sufficiently high throughput, thus a hybrid solution becomes unnecessary.

There are also many generic load balancing software packages, the most popular of which are NGINX [14], HAProxy [7], and Linux Virtual Server [11]. They usually run on single servers, but it is also possible to deploy multiple servers in an ECMP group behind a router to achieve the scale-out model. They all provide consistent hashing mechanisms. Compared to Maglev, they mostly prioritize minimum disruption over even load balancing as is done by [28] and [38]. Because they are designed for portability, they are not aggressively optimized for performance.

Consistent hashing [28] and rendezvous hashing [38] were originally introduced for the purpose of distributed cache coordination. Both methods provide guaranteed resilience such that when some backends are removed, only table entries pointing to those backends are updated. However, they don't provide good load balancing across backends, which is an essential requirement for load balancers. On the contrary, Maglev's consistent hashing method achieves perfect balance across the backends at the cost of slightly reduced resilience, which works well in practice when paired with connection tracking. Another option for implementing consistent hashing is distributed hash tables such as Chord [37], but this would add extra latency and complexity to the system.

Some of the performance optimization techniques used in Maglev have been extensively studied since 1990s. Smith et al [36] suggested to improve application throughput by reducing interrupts and memory copying. Mogul et al [33] developed a polling-based mechanism to avoid receive livelock caused by interrupts. Edwards et al [19] explored the idea of userspace networking but did not manage to bypass the kernel completely. Marinos et al [31] showed that specialized userspace networking stacks with kernel bypass can significantly improve application throughput. Hanford et al [25] suggested to distribute packet processing tasks across multiple CPU cores to improve CPU cache hit ratio. CuckooSwitch [41] is a high-performance software L2 switch. One of its key techniques is to mask memory access latency through batching and prefetching. RouteBricks [18] explained how to effectively utilize multi-core CPUs for parallel packet processing.

Several kernel bypass techniques have been developed recently, including DPDK [8], OpenOnload [15], netmap [35], and PF_RING [17], etc. A good summary of popular kernel bypass techniques is presented in [10]. These techniques can be used to effectively accelerate packet processing speed, but they all come with certain limitations. For example, DKPK and OpenOnload are tied to specific NIC vendors while netmap and PF_RING both require a modified Linux kernel. In Maglev we implement a flexible I/O layer which does not require kernel modification and allows us to conveniently switch among different NICs. As with other techniques, Maglev takes over the NIC once started. It uses the TAP interface to inject kernel packets back to the kernel.

GPUs have recently started becoming popular for high-speed packet processing [24, 39]. However, Kalia et al [27] recently showed that CPU-based solutions are able to achieve similar performance with more efficient resource utilization if implemented correctly.

7 Conclusion

This paper presents Maglev, a fast, reliable, scalable and flexible software network load balancer. We built Maglev to scale out via ECMP and to reliably serve at 10Gbps line rate on each machine, for cost-effective performance with rapidly increasing serving demands. We map connections consistently to the same backends with a combination of connection tracking and Maglev hashing. Running this software system at scale has let us operate our websites effectively for many years, reacting quickly to increased demand and new feature needs.

Acknowledgements

We are grateful to Adam Lazur, Alex Tumko, Amin Vahdat, Angus Lees, Aspi Siganporia, Ben Treynor, Bill Coughran, Brad Calder, Craig Bergstrom, Doug Orr, Dzevad Trumic, Elliott Karpilovsky, Jeff Mogul, John T. Reese, Kyle Moffett, Luca Bigliardi, Mahesh Kallahalla, Mario Fanelli, Mike Dalton, Mike Shields, Natalya Etina, Nori Heikkinen, Pierre Imai, Roberto Peon, Simon Newton, Tina Wong, Trisha Weir, Urs Hölzle, and many others for their significant contributions to this paper and the success of Maglev. We would also like to thank our shepherd Nathan Bronson and the anonymous reviewers for their insightful feedback.

References

[1] A10. http://www.a10networks.com.

[2] Array networks. http://www.arraynetworks.com.

[3] Barracuda. http://www.barracuda.com.

[4] Embrane. http://www.embrane.com.

[5] F5. http://www.f5.com.

[6] Google cloud platform. http://cloud.google.com.

[7] Haproxy. http://www.haproxy.org.

[8] Intel dpdk. http://www.dpdk.org.

[9] Kemp. http://www.kemptechnologies.com.

[10] Kernel bypass. http://blog.cloudflare.com/kernel-bypass.

[11] Linux virtual server. http://www.linuxvirtualserver.org.

[12] Load balancer .org. http://www.loadbalancer.org.

[13] Netscaler. http://www.citrix.com.

[14] Nginx. http://www.nginx.org.

[15] Openonload. http://www.openonload.org.

[16] F. Chen, R. K. Sitaraman, and M. Torres. End-user mapping: Next generation request routing for content delivery. In *Proceedings of SIGCOMM*, 2015.

[17] L. Deri. Improving passive packet capture: Beyond device polling. In *Proceedings of SANE*, 2004.

[18] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of SOSP*, 2009.

[19] A. Edwards and S. Muir. Experiences implementing a high performance tcp in user-space. In *Proceedings of SIGCOMM*, 1995.

[20] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. Edinburgh: Oliver and Boyd, 1963.

[21] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of NSDI*, 2007.

[22] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of SIGCOMM*, 2014.

[23] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of SIGCOMM*, 2011.

[24] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of SIGCOMM*, 2010.

[25] N. Hanford, V. Ahuja, M. Balman, M. K. Farrens, D. Ghosal, E. Pouyoul, and B. Tierney. Characterizing the impact of end-system affinities on the end-to-end performance of high-speed flows. In *Proceedings of NDM*, 2013.

[26] V. Jacobson and B. Felderman. Speeding up networking. http://www.lemis.com/grog/Documentation/vj/lca06vj.pdf.

[27] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using gpus in software packet processing. In *Proceedings of NSDI*, 2015.

[28] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of ACM Symposium on Theory of Computing*, 1997.

[29] C. Labovitz. Google sets new internet record. http://www.deepfield.com/2013/07/google-sets-new-internet-record/.

[30] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. Internet inter-domain traffic. In *Proceedings of SIGCOMM*, 2010.

[31] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *Proceedings of SIGCOMM*, 2014.

[32] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *Proceedings of USENIX ATC*, 2010.

[33] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proceedings of USENIX ATC*, 1996.

[34] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of SIGCOMM*, 2013.

[35] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proceedings of USENIX Security*, 2012.

[36] J. Smith and C. Traw. Giving applications access to gb/s networking. *Network, IEEE*, 7(4):44–52, 1993.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM*, 2001.

[38] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, 1998.

[39] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. Multi-layer packet classification with graphics processing units. In *Proceedings of CoNEXT*, 2014.

[40] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of SoCC*, 2010.

[41] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of CoNEXT*, 2013.

Enabling ECN in Multi-Service Multi-Queue Data Centers

Wei Bai[1], Li Chen[1], Kai Chen[1], Haitao Wu[2]
[1]*SING Group @ HKUST* [2]*Microsoft*

Abstract

Recent proposals have leveraged Explicit Congestion Notification (ECN) to achieve high throughput low latency data center network (DCN) transport. However, most of them implicitly assume each switch port has one queue, making the ECN schemes they designed inapplicable to production DCNs where multiple service queues per port are employed to isolate different traffic classes through weighted fair sharing.

In this paper, we reveal this problem by leveraging extensive testbed experiments to explore the intrinsic tradeoffs between throughput, latency, and weighted fair sharing in multi-queue scenarios. Using the guideline learned from the exploration, we design MQ-ECN, a simple yet effective solution to enable ECN for multi-service multi-queue production DCNs. Through a series of testbed experiments and large-scale simulations, we show that MQ-ECN breaks the tradeoffs by delivering both high throughput and low latency simultaneously, while still preserving weighted fair sharing.

1 Introduction

Data centers host a variety of applications and services with diverse network requirements: some services (*e.g.*, monitoring services) demand low latency for sporadic short messages; some (*e.g.*, data-parallel computation [13]) require high throughput for large flows; while others (*e.g.*, online data-intensive applications) desire both high throughput and low latency for a large number of concurrent flows [18].

To meet these requirements, ECN has been employed as a powerful tool by recent DCN transport proposals such as [6, 8, 29, 31, 32], and they show that a properly tuned ECN marking scheme can deliver high throughput and low latency simultaneously [31]. Due to their simplicity and effectiveness, ECN-based transports such as DCTCP [6] and DCQCN [32] are widely used in industry—DCTCP has been integrated into various OS kernels [3, 4] and deployed in DCNs of Google [27] and Morgan Stanley [19]; while DCQCN has been deployed in DCNs of Microsoft [32] to enable RDMA.

A further look at these proposals reveals that their ECN marking schemes are mostly designed based on the implicit assumption that each switch port only has one queue. However, the industry trend in production DCNs is going beyond such one queue per port paradigm [8, 9]. Today's commodity switches already support 4–8 classes of service queues per port [9, 10, 20]. Current operation practice is to leverage queues to segregate traffic from different services and enforcing weighted fair sharing among different queues [8, 9, 19]. For example, operators assign a higher weight to all traffic belonging to a more important *real-time* search application over a *background* backup application, thus providing differentiated network performance. A key question in such single-queue to multi-queue transition is the applicability of ECN, which remains unexplored.

We point out, via extensive testbed experiments, that the prior ECN schemes developed for the single queue model fall short when directly migrated to the multi-queue scenarios (§2). There exist fundamental tradeoffs between high throughput, low latency, and weighted fair sharing. Our experiments demonstrate: 1) applying per-queue ECN with the standard marking threshold derived before on each queue independently ensures high throughput, but can incur high latency when many queues are active; while apportioning this threshold among all the queues statically according to their weights guarantee low latency, but can degrade throughput when few queues are live; 2) applying per-port ECN with such standard threshold can maintain both high throughput and low latency, but violating weighted fair sharing across different queues.

Motivated by above problem, we seek a solution that can break the tradeoffs and enable ECN for multi-service multi-queue DCNs. To this end, we present MQ-ECN, a simple yet effective solution that achieves our goal (§3). First, MQ-ECN takes the per-queue ECN approach to preserve weighted fair sharing. Then, at its heart, MQ-ECN adjusts the ECN marking threshold for each queue based on its dynamic weighted fair share rate, rather than sticking to its static fair share weight, which enables MQ-ECN to well adapt to traffic variations while maintaining both high throughput and low latency in a highly dynamic DCN environment.

We explain that MQ-ECN is feasible to implement on existing commodity switch hardware as MQ-ECN just requires one additional moving average register per port compared to the standard ECN/RED switch implementation (§4.1). We also present a MQ-ECN software implementation for testbed evaluation (§4.2). In our software prototype, MQ-ECN is implemented as a Linux `qdisc` kernel module running on a multi-NIC server to emulate switch behaviors.

We build a small-scale testbed with 9 Dell servers connected to a 9-port server-emulated MQ-ECN-enabled switch. We evaluate the basic properties of MQ-ECN on the testbed using realistic workloads [6, 16, 25]. Our experiments demonstrate that MQ-ECN achieves both high throughput and low latency simultaneously, while strictly preserving weighted fair sharing. For example, compared to per-queue ECN with the standard threshold, MQ-ECN achieves up to 72.8% lower 99th percentile FCT for small flows while delivering similar performance (*e.g.*, within 2.7%) for large flows (§5.1).

To complement small-scale testbed experiments, we conduct large-scale ns-2 [5] simulations to deep-dive into MQ-ECN. Our simulation results further confirm the superior performance of MQ-ECN. For example, compared to per-queue ECN with the standard threshold, MQ-ECN reduces the 99th percentile FCT for small flows by up to 43.7%. In addition, MQ-ECN achieves up to 13.2% lower average FCT for large flows compared to per-queue ECN with the minimum threshold (§5.2). Finally, we show, through a series of targeted simulations, that MQ-ECN is robust to different network environments and parameter settings, such as the number of queues, queue weights, transport protocols, and so on (§5.3).

To make our work easy to reproduce, we make our codes available online at: `http://sing.cse.ust.hk/projects/MQ-ECN`.

2 Problem Exploration

In this section, we begin by introducing the ECN mechanisms supported by existing commodity switching chips. Then, we explore the problems and tradeoffs of applying ECN in multi-service multi-queue DCNs. Finally, we summarize our design goals.

2.1 ECN on Commodity Switching Chips

Today's commodity switching chips provide multiple ECN/RED configuration options. For example, in our testbed, the Broadcom BCM-56538 chip supports per-queue, per-port, and per service pool ECN markings. For all schemes, the marking decision is made when a packet is enqueued (required by RED [15]). The main difference among them is that they use buffer occupancy in different egress entities to make marking decisions.

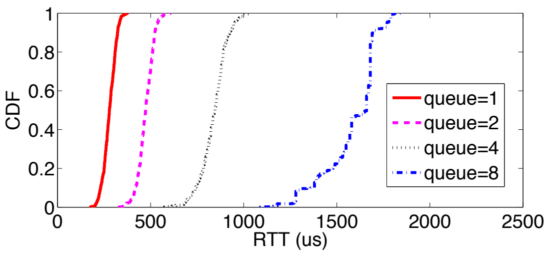


Figure 1: [Testbed] RTT: under per-queue ECN with the standard threshold, more queues lead to worse latency.

Briefly, in per-queue ECN marking, each queue has its own threshold and performs ECN marking independently to other queues. In per-port ECN marking, each port is assigned a single marking threshold. When the sum of queue buffer occupancy belonging to the same port is larger than the marking threshold, packets will get ECN marking. In per service pool ECN marking, packets are marked when total buffer occupancy in a shared buffer pool exceeds the marking threshold.

2.2 Problems and Tradeoffs

Before exploring the problems, we first introduce the *standard* ECN marking threshold derived by prior works [7, 31] based on the single queue model. Consider synchronized flows with identical round-trip times sharing the only queue of a bottleneck link, according to [7, 31], to fully utilize the link bandwidth while achieving low latency, the ECN marking threshold K[1] should be set as follows:

$$K = C \times RTT \times \lambda \tag{1}$$

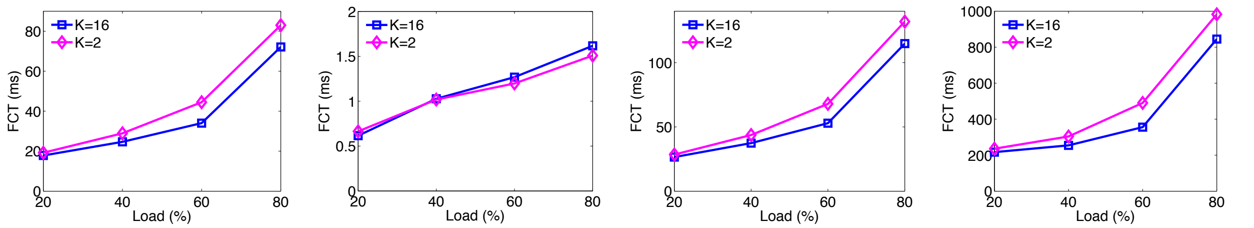
where RTT is average round-trip time, C is link capacity, and λ is a tunable parameter closely related to congestion control algorithms[2]. In production DCNs, round-trip times are relatively stable and operators can estimate RTT through large-scale measurements to compute the standard ECN marking threshold [17, 31].

2.2.1 Per-queue ECN with the standard threshold

In multi-queue environment, per-queue ECN marking is widely employed by operators for its good isolation among different queues. However, how to set the ECN marking threshold for each queue is a challenge. DCN traffic is well known for its volatility and burstiness [11, 16]. Thus, to achieve high utilization in any

[1] We are aware that ECN/RED has two (low and high) thresholds. Many ECN-based transports [6, 29, 31] set them to the same value. Without loss of generality, we also assume that the low and high thresholds are set to the same value to simplify analysis.

[2] For example, $\lambda = 1$ for regular ECN-enabled TCP which simply cuts window by half in the presence of ECN [31].



(a) Overall (b) (0,100KB] (c) (100KB,10MB] (d) (10MB,∞)

Figure 2: [Testbed] Average FCT statistics: under per-queue ECN with minimum threshold (*e.g.*, K=2), it suffers from degraded throughput as only one queue is active, thus leading to higher FCT overall.

condition, some network operators have configured the standard marking threshold, *e.g.*, $C \times RTT \times \lambda$, for each queue. Under this configuration, any queue can fully utilize link capacity independently. However, the problem is that when N queues are busy simultaneously, the total buffer occupancy can easily reach N times the standard threshold, thus introducing high queueing delay and huge buffer pressure[3].

To quantify such impairment, we build a small testbed consisting of 15 servers connected to a Pica8 P-3295 GbE switch. DCTCP is enabled on all the servers[4]. We configure Deficit Weighted Round Robin [26] with equal quantum per queue on the switch. We set the per-queue ECN marking threshold to 16 packets. We generate 8 long-lived flows using `iperf` from 8 servers to the same receiver. We vary the number of queues from 1 to 8 and evenly classify all the flows into these queues by setting different Differentiated Services Code Point (DSCP) values. Given all the queues should have similar queueing delay, we run `ping` in an active queue to measure RTT. Figure 1 shows RTT distributions. Clearly, more queues lead to worse latency. Compared to the single queue, the average and 99th percentile RTTs achieved by 8 queues degrade by 5.7X (279μs to 1582μs) and 4.9X (375μs to 1850μs), respectively.

Observation 1: *Per-queue ECN with the standard threshold suffers from poor latency when many queues are concurrently active.*

2.2.2 Per-queue ECN with the minimum threshold

To address the defect above, a natural way is to apportion the standard threshold among all the queues according to their fair share weights. Assume each port has N queues in total and the weight of queue i is W_i, then the *minimum* threshold for queue i, K_i, can be set as:

$$K_i = \frac{W_i}{\sum_{j=1}^{N} W_j} \times C \times RTT \times \lambda \qquad (2)$$

Note that $\frac{W_i}{\sum_{j=1}^{N} W_j}$ is the normalized weight for queue i and $\frac{W_i}{\sum_{j=1}^{N} W_j} \times C$ is the minimum guaranteed rate for this queue. Hence, the minimum threshold K_i ensures that each queue can receive its minimum guaranteed bandwidth. Since K_i is proportional to W_i, it can also preserve weighted fair sharing among different queues. Furthermore, given that $\sum_{j=1}^{N} K_j = C \times RTT \times \lambda$, such configuration can achieve good latency and burst tolerance regardless of the total number of queues.

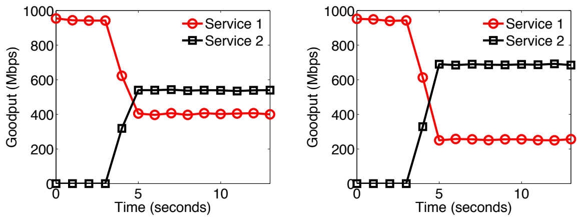
However, the problem of this method is that it can seriously degrade link utilization, especially when only few queues are active. The reason is that the bandwidth for inactive queues cannot be fully utilized by active queues as they are throttled by the statically-configured minimum ECN marking thresholds. The low throughput directly degrades the flow completion times (FCT).

To quantify this impact, we develop a client/server application to generate traffic according to the web search workload [6]. The client instance, running on one server, periodically generates requests to server instances, running on the other 14 machines, to fetch data. All the traffic is classified into the same switch queue. Since only one queue is active, to fully utilize link capacity, we should assign the standard threshold, *e.g.*, 16 packets, for this queue. Given we have 8 queues with equal weights, the corresponding minimum threshold for one queue is 2 packets. Thus, we evaluate the performance of both 16 packets and 2 packets in the experiment. Figure 2 shows the FCT results across different flow size regions. It turns out that the scheme with the threshold of 16 packets achieves $7.2-23.5\%$ lower overall average FCT (due to higher throughput) compared to that with the minimum threshold of 2 packets. This performance improvement stems mainly from the flows larger than 100KB.

Observation 2: *Per-queue ECN with the minimum threshold cannot maintain high throughput especially when few queues are concurrently active.*

[3] Taking Pica8 P-3922 10GbE switch [1] as an example, it has 9MB buffer shared by 384 queues (48 ports×8 queues/port). DCTCP [6] recommends using at least 65 packets as the threshold for 10G networks. Hence, when 97 queues are busy simultaneously, the buffer is likely to be overfilled. Frequent packet drops can also severely degrade latency.

[4] By default, we choose DCTCP as the transport protocol for all experiments/simulations in this paper except special declaration.



(a) 1 flow : 4 flows

(b) 1 flow : 8 flows

Figure 3: [Testbed] Aggregate goodput statistics: in per-port ECN, a queue with more flows grabs more bandwidth.

2.2.3 Per-port and per service pool ECN

Unlike above two approaches, per-port ECN can achieve both high throughput and low latency with the standard threshold. However, the problem is that it cannot ensure isolation among different queues of the same port [10]. This is because packets from one queue may get marked due to buffer occupancy of the other queues belonging to the same port. This undesirable interaction can severely violate weighted fair sharing among queues. We believe the problem will deteriorate under per service pool ECN marking because in such case queues belonging to even different ports may interfere with each other.

To understand this impairment, we start several long-lived TCP flows from two senders to the same receiver. We classify traffic into two services based on their senders. On the switch, both services have a equal-quantum dedicated queue. The per-port marking threshold is 16 packets. We vary the numbers of flows and measure aggregate goodputs for two services. Ideally, both services should always equally share the link capacity. Figure 3 shows the actual share results. When service 1 has 1 flows and service 2 has 4 flows, their aggregate goodputs are 403Mbps and 539Mbps, respectively. When the number of flows in service 2 is increased to 8, its aggregate goodput further reaches 688Mbps. This suggests that under per-port ECN marking, packets of service queue 1 get over-marked due to the aggressiveness of packets in service queue 2, thus making service 1 fail to achieve its weighted fair share rate.

Observation 3: *Per port and per service pool ECN can violate weighted fair sharing among different queues.*

2.3 Design Goals

Motivated by the above problems, we aim to design an ECN marking scheme for multi-service multi-queue production DCNs with the following properties:

- **High throughput:** Our scheme must be work-conserving. Active services should be able to fully utilize the network bandwidth as long as they have enough demands.
- **Low latency:** Our scheme should maintain low buffer occupancy in order to provide low queueing delay and good burst tolerance.
- **Weighted fair sharing:** Our solution should strictly preserve the weighted fair sharing policy among different service queues at any time.
- **Compatible with legacy ECN/RED implementation:** Although there are a few ECN improvements that leverage dequeue marking [31], to the best of our knowledge, few chip providers have offered the support. Therefore, we choose to design a scheme that can benefit from most ECN features that are available on existing switching commodity chips.

We show how MQ-ECN achieves the first three goals in the next section. To achieve the last goal, we require MQ-ECN to perform RED-like enqueue marking, *e.g.*, comparing the average queue length against a threshold at the enqueue side. And we discuss our implementation requirements in §4.1.

3 The MQ-ECN Design

3.1 Design Guideline

The above problem exploration has guided our design of MQ-ECN. The lesson we learned is two-fold:

- To avoid interference among different queues and preserve weighted fair sharing, the ECN marking should be performed on a per queue basis while complying with the weights across different queues.
- To achieve both high throughput and low latency simultaneously, we should not set static ECN marking thresholds for queues—applying the standard threshold on each queue independently can cause high latency (observation 1), while apportioning this threshold among all queues statically according to their weights can lead to low throughput (observation 2). Instead, the ECN marking threshold for each queue should adapt to traffic dynamics, and it should be set in a way that can *barely* maintain its weighted fair share rate while not introducing extra queueing delay. More specifically, for each queue, if its input rate is larger than its weighted fair share rate, we should use ECN to properly throttle it for latency; otherwise, we should not impose any constraint in order not to affect its throughput. As a result, the core of MQ-ECN is to derive such a proper ECN marking threshold for each queue according to its weighted fair share rate.

In our implementation, we find that the ECN threshold setting is closely related to the underlying packet scheduler that enforce the weighted fair sharing. Thus, in the following, we first describe the base design of MQ-ECN with ideal Generalized Processor Sharing (GPS) [23].

Then we discuss how to extend the base design to practical packet scheduling algorithms that are widely implemented in existing commodity switching chips.

3.2 MQ-ECN with Ideal GPS

3.2.1 Base Model

We consider a switch output link with the capacity C. The switch uses GPS as the underlying scheduler. There are N flows in total and the demand of flow i is r_i. Flow i is mapped to queue i with the weight of w_i[5]. The total demand is A, and $A = \sum_{i=1}^{N} r_i$. Let α denote the fair share rate and $w_i\alpha$ is the corresponding weighted fair share rate for flow i. According to [28], if $A > C$, the link is congested and α is the unique solution for equation $C = \sum_{i=1}^{N} min(r_i, w_i\alpha)$; if $A \leq C$, then there is no congestion and $\alpha = max\{\frac{r_i}{w_i}\}$. The output rate of queue i is given by $min(r_i, w_i\alpha)$.

According to design guideline in §3.1, we classify the queues into two categories based on the relations between their input rates and weighted fair share rates. For queues whose $r_i > w_i\alpha$, we use the following Equation (3) to throttle their input rates to keep queue occupancy and maintain low latency. For queues whose $r_i \leq w_i\alpha$, they should not be constrained.

$$K_i = w_i\alpha \times RTT \times \lambda \tag{3}$$

However, in order to enforce the scheme, the premise is to identify the queues whose $r_i > w_i\alpha$ and estimate their weighted fair share rates (*e.g.*, output rates) $w_i\alpha$, which was a challenge. Some previous works [22, 28] first estimate the input rates r_i and then calculate the weighted fair share rates $w_i\alpha$ using various complicated heuristics in the context of FIFO scheduling. Accurate rate estimation is challenging in data centers as traffic is volatile and bursty. Unlike previous approaches, we use GPS as the underlying scheduler. Thus, we can take advantage of the special properties of GPS to address this challenge in a much simpler way.

Note that GPS serves all backlogged queues in a bit-by-bit round-robin fashion. Assume $quantum_i = w_i$ bits is the quantum for queue i in a round, and T_{round} is the total time to serve all queues once. In case $r_i > w_i\alpha$ for queue i, the data in the queue will keep growing and eventually use up its $quantum_i$ in a round, then we can use $\frac{quantum_i}{T_{round}}$ to calculate the output rate of queue i, which is $w_i\alpha$. Thus, Equation (3) can be translated to:

$$K_i = \frac{quantum_i}{T_{round}} \times RTT \times \lambda \tag{4}$$

where $quantum_i$, RTT and λ are known while T_{round} can be well estimated through continuous sampling as we show later in §3.3.

Interestingly, though intended for queues whose $r_i > w_i\alpha$, we find that Equation (4) can also be applied to queues whose $r_i \leq w_i\alpha$ with no harm. Here is the reason. For a queue i whose $r_i \leq w_i\alpha$, the data drained in a round is no more than $quantum_i$, this means that we can use $\frac{quantum_i}{T_{round}}$ to safely cap the output rate of queue i, which is r_i. Thus applying Equation (4) to queue i does not throttle its input rate, but still allows it to grow beyond its weighted fare rate before taking effect. This greatly simplifies our design because, we can apply Equation (4) to every queue with no differentiation, without the need of explicitly identifying the relations between their input rates and weighted fair share rates. As a result, Equation (4) establishes the ECN marking scheme of MQ-ECN with the ideal GPS scheduler.

3.2.2 Why it works?

We find Equation (4) well achieves our design goals in §2.3. First, $quantum_i$ ensures that different queues have thresholds in proportion to their weights, thus preserving the weighted fair sharing. Second, T_{round} reflects traffic dynamics of the link and automatically balances its throughput and latency. When there are more queues whose input rates exceed their weighted fair share rates, T_{round} tends to become larger, then K_i automatically becomes smaller to maintain low latency. When there are fewer queues reach their weighted fair share rates, T_{round} becomes smaller, then K_i automatically becomes larger to maintain high throughput.

Furthermore, in practice, T_{round} may change drastically because data center traffic is volatile and bursty. Accurately estimating T_{round} is challenging and deviation is unavoidable. However, we find that MQ-ECN can be self-healing:

- Assume that T_{round} is over-estimated initially, we get a smaller K_i which degrades throughput. Then more and more queues will be over-throttled by MQ-ECN and cannot achieve their weighted fair share rates. As a consequence, T_{round} becomes smaller.
- Assume that T_{round} is under-estimated initially, we get a larger K_i which increases latency. Then more and more queues will ramp up and exceed their weighted fair share rates. As a consequence, T_{round} becomes larger.

Therefore, an inaccurate initial estimation for T_{round} can be cured by itself eventually in the later stages. Furthermore, in our implementation, to prevent any temporary impact of under-estimation, we use Equation (5) below to bound it, considering that the weighted fair share rate should never be larger than the link capacity. Our evaluation results in §5 further confirm that MQ-ECN works well in practice.

$$K_i = min(\frac{quantum_i}{T_{round}}, C) \times RTT \times \lambda \tag{5}$$

[5] Given each queue only has one flow, we use 'flow' and 'queue' interchangeably in §3.

3.3 MQ-ECN with Practical Packet Schedulers

In this section, we show how to extend the solution derived from ideal GPS to practical packet scheduling algorithms that try to approximate GPS. These schemes can be generally divided into two classes: fair queueing and round robin. Fair queueing schemes, such as Weighted Fair Queueing [14], achieve good fairness in general, but they are expensive to implement due to high $O(log(n))$ time complexity (n is the number of queues). Round robin schemes [21, 26] suffer from short-term burstiness and unfairness, but they are widely implemented in commodity switching chips for their $O(1)$ time complexity. For example, some dominant chipsets such as Broadcom Trident-I&II [2] adopted in many production data centers only support several round-robin schemes, such as Weighted Round Robin (WRR) and Deficit Weighted Round Robin (DWRR). Hence, we mainly focus on round robin schemes in this paper.

To apply Equation (5) for a packet scheduling algorithm, we need to obtain average round time T_{round} and per-queue quantum $quantum_i$ in the new context of this algorithm. Here, we show how to get T_{round} and $quantum_i$ for two popular round-robin schemes: WRR and DWRR. We envision that similar approaches can be extended to other round-robin schemes.

3.3.1 Estimate T_{round}

Round-robin packet scheduling algorithms usually serve queues in a circular order. Intuitively, one can obtain a sample of T_{round} whenever the scheduler finishes serving all the queues in a round. However, the sample frequency of this approach is directly affected by the total number of active queues. When many queues are concurrently active, it cannot track traffic dynamics efficiently. Hence, we propose to sample T_{round} whenever a queue just finishes its service in a round. A benefit is that such sampling frequency is independent to the number of queues. We assume that each queue i maintains a variable T_{pre} to store the time stamp when queue i finishes the service in previous round. Every time queue i finishes its service, it records the current time stamp T_{now} and calculates a round time sample T_{sample} as: $T_{sample} = T_{now} - T_{pre}$. Then we reset T_{pre} with T_{now}. Every time we get a sample, we smooth T_{round} using exponential filter as follows:

$$T_{round} = \beta \times T_{round} + (1-\beta) \times T_{sample} \tag{6}$$

where β is a parameter in $(0, 1)$. We note that the above approach may have two potential limitations. First, T_{round} will be updated too frequently when there are many empty queues. Second, sampling stalls when the link is idle. To address the first limitation, we only sample T_{round} on active queues. If queue i is empty, we just reset T_{pre} with T_{now} and move forward to next queue. To address the second limitation, we simply set T_{round} as $\beta \times T_{round}$ (as if we get a T_{sample} of 0) when the switch port is idle for a pre-defined T_{idle} time.

3.3.2 Derive $quantum_i$

Deriving $quantum_i$ for WRR and DWRR is relatively simple. Recall that $quantum_i$ defines the maximum amount of data a queue can send in a round.

- **WRR**: In the latest implementation of WRR on chips, each queue is configured with a quantum Q_i worth of bits, and in each round queue i can at most transmit Q_i (rather than a fixed number of packets in earlier proposals [21]). Thus, $quantum_i = Q_i$ for WRR.
- **DWRR**: In the implementation of DWRR, each queue is also configured with a quantum Q_i worth of bits. Typically, Q_i should be no smaller than maximum transmission unit (MTU) to provide $O(1)$ time complexity [26]. Instead of Q_i, the DWRR scheduler maintains a *deficit counter* for each queue to bound the maximum amount of data to send in each round. This deficit counter maintains the unused quota left in previous round, and is incremented by Q_i in current round (or reset to 0 if the queue is empty). Considering queue i keeps backlogged for M rounds, let $sent_i$ denote the total amount of bits sent by queue i in this period. We can bound $sent_i$ as follows [26]:

$$M \times Q_i - MTU \leq sent_i \leq M \times Q_i + MTU \tag{7}$$

On average, the amount of data queue i can send in each round is: $\frac{sent_i}{M} = [Q_i - \frac{MTU}{M}, Q_i + \frac{MTU}{M}] \approx Q_i$. Thus, we set $quantum_i = Q_i$ for DWRR.

3.4 Discussion

Weighted Fair Queueing: The reader may wonder how to extend MQ-ECN to weighted fair queueing (WFQ) or other fair queueing schemes. Unlike round robin schemes, WFQ does not have the explicit round concept. So it is difficult to directly apply Equation (5) to WFQ.

A straightforward approach is to divide the standard ECN marking threshold to all backlogged queues. For example, we can define a queue is backlogged if it is not empty. We use w_{sum} to denote the sum of weights of all backlogged queues, and w_{sum} can also be updated using exponential filter like T_{round}. The ECN marking threshold for queue i can be set as $\frac{w_i}{w_{sum}} \times C \times RTT \times \lambda$. However, this formula is built on an implicit assumption that a non-empty queue is able to use up its weighted fair share rate, which may not always hold in all cases. In fact, $\frac{w_i}{w_{sum}} \times C$ is the lower bound for $w_i\alpha$ when link is congested. Hence, the above approach may underestimate weighted fair share rates and derive lower ECN marking thresholds. We believe a key factor for this approach is to accurately define the backlogged queues and quickly identify them, which may not be so easy. However, a more general MQ-ECN scheme that better supports WFQ is our future work.

Strict Priority Queueing: In production DCNs, network operators may reserve few (typically one) extra queues with strict higher priority to deliver a small amount of important control messages. The remaining queues are typically scheduled using DWRR/WRR in the lowest priority. MQ-ECN does not directly apply to such scenario as well. As an approximation, for strict higher priority queues without round concept, we statically set their marking thresholds to the standard marking threshold $C \times RTT \times \lambda$. For DWRR/WRR queues in the lowest priority, we still apply MQ-ECN (Equation (5)) to calculate dynamic marking thresholds.

Probabilistic Marking: ECN/RED actually has two thresholds and a maximum marking probability to perform a probabilistic marking. When two thresholds are set to the same value, the maximum marking probability no longer takes effect. Some transports, *e.g.*, DCQCN [32], require such probabilistic marking by setting different values for two thresholds. MQ-ECN can be easily extended to perform such probabilistic marking. Let K_{min}, K_{max} and P_{max} denote the low standard threshold, the high standard threshold and the maximum marking probability derived under the single queue model. The low/high thresholds and the maximum probability for queue i of MQ-ECN are $K_{i,min}$, $K_{i,max}$ and $P_{i,max}$, respectively. They can be given as: $K_{i,min} = K_{min} \times min(\frac{quantum_i}{C \times T_{round}}, 1)$, $K_{i,max} = K_{max} \times min(\frac{quantum_i}{C \times T_{round}}, 1)$, and $P_{i,max} = P_{max}$.

4 Implementation

In this section, we first analyze the feasibility of MQ-ECN implementation on switching chips, and then describe a software prototype of MQ-ECN in detail. An implementation of MQ-ECN on switching chip hardware is under negotiation but beyond the scope of this work.

4.1 Switch Implementation

In typical switch implementation for ECN/RED, there is a comparison for an averaged queue length and a static threshold, which is setup using registers. In MQ-ECN's implementation, the comparison is between the same average queue length and a dynamic threshold. In this section, we discuss the implementation complexity for the dynamic threshold. To calculate K_i for a queue, we need to calculate T_{round}. The calculation of T_{round} can be implemented by the moving average of round robin time taken on scheduling. Compared to the per queue average queue length, T_{round} is per port. Therefore, MQ-ECN keeps the same scale implementation complexity as ECN/RED, as we just need one additional register per port to store T_{round}.

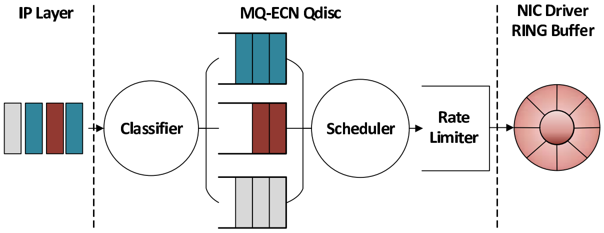


Figure 4: MQ-ECN software stack.

However, one potential challenge is that too frequent moving average calculation cannot be easily achieved by switching chips. This problem becomes more and more serious as the link capacity of production DCNs keeps increasing in recent years. To the best of our knowledge, some chip vendors take a time based moving average calculation to address this challenge. For example, for average queue length calculation of ECN/RED, the moving average is taken for a static interval at microseconds granularity rather than each packet arrival/departure. Similarly, our T_{round} moving average update can also be implemented using a time based version. In our discussion, we prefer a time related to the transmission time of an MTU. Taking 10G link capacity and MTU=9KB as an example, the transmission time is 7.2μs. The moving average calculation at this time granularity can be achieved by most switching chip vendors as we know.

In summary, MQ-ECN maintains the same scale implementation complexity as ECN/RED as it just requires one addition moving average register per port.

4.2 Software Prototype

Since we cannot program our switching chips, we use a server with multiple Network Interface Cards (NICs) to emulate the switch and implement MQ-ECN on top of that. MQ-ECN is implemented as a new Linux queueing discipline (`qdisc`) kernel module. Hence, we can avoid the overhead of data copy and context switch between user and kernel space. Figure 4 shows the software stack of MQ-ECN. MQ-ECN prototype has three components: a packet classifier, a packet scheduler, and a rate limiter. Instead of modifying the Linux `tc`, we expose new `sysctl` interfaces for users to configure the new `qdisc` module in user space.

Packet Classifier: MQ-ECN kernel module maintains multiple FIFO transmit queues. Packets are classified into different queues based on the IP DSCP field. When

MQ-ECN kernel module receives a packet from IP layer, it: 1) classifies the packet based on DSCP value, 2) calculates the ECN marking threshold of the corresponding queue, 3) performs ECN marking if needed, and 4) enqueues the packet.

Packet Scheduler: The packet scheduler of MQ-ECN kernel module is built on the top of the DWRR scheduler available in Linux. Our implementation can also be easily extended to WRR by resetting deficit counter to 0 whenever a queue finishes its service in a round.

The DWRR scheduler maintains a linked list for all active queues. When an empty queue receives a packet, it is inserted to the tail of the linked list. The scheduler always serves the head node of the linked list. If a queue just finishes its service and but still has packets, it is inserted to the tail of the linked list again. Each queue has a variable T_{start} to store the time stamp when this queue is inserted to the linked list last time. Whenever a queue finishes its service, we use current time minus T_{start} to get a sample of T_{round}. In this way, we only sample T_{round} on active queues, just as §3.3.1 described.

Rate Limiter: One implementation challenge is to make the buffer occupancy in `qdisc` reflect the real buffer occupancy of the emulated switch port. A packet dequeued by `qdisc` further goes through NIC driver and NIC hardware before it is delivered to the wire. If we dequeue packets from `qdisc` as fast as possible, many packets can still get queued on NIC driver and hardware. Consequently, the buffer occupancy in `qdisc` is likely to be smaller than the actual buffer occupancy of the emulated switch port. To avoid such impact, we implement a Token Bucket rate limiter to rate-limit the outgoing traffic from `qdisc` at 99.5% of the line rate. The bucket size is ∼1.67 MTU (2.5KB) in our experiment, which is large enough to saturate more than 99% of link capacity while introducing little burstiness. In this way, we can eliminate undesirable buffering in other places and make the buffer occupancy in `qdisc` accurately reflect the buffer occupancy of the emulated switch port.

To confirm the effectiveness of the rate limiter, we install MQ-ECN kernel module on a server with 10 GbE NICs to emulate switch. Two other servers are connected to this software switch. The shaping rate is 995Mbps. The bucket size is 2.5KB. We start a long-lived TCP flow to measure goodput. The goodputs with and without kernel module are 937Mbps and 942Mbps, respectively. MQ-ECN module introduces ∼0.53% goodput degradation, exactly enforcing the desired rate (995Mbps).

5 Evaluation

In this section, we use testbed experiments and ns-2 [5] simulations to answer following three key questions:

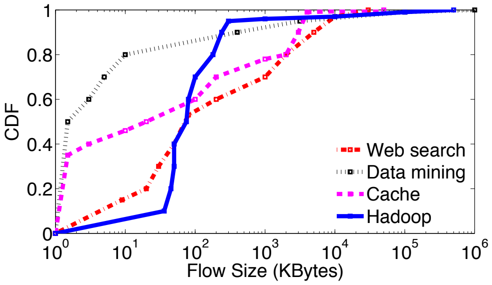
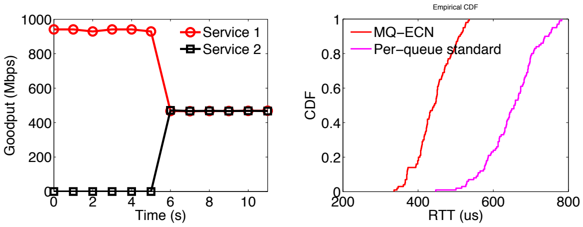


Figure 5: Flow size distributions used for evaluation.

- **How does MQ-ECN perform in practice?** In a static flow experiment (§5.1.1), we show that MQ-ECN strictly preserves weighted fair sharing while maintaining high throughput and low latency. Using realistic workloads in our testbed experiments (§5.1.2), we show that MQ-ECN outperforms the other schemes. For example, it achieves up to 72.8% lower 99th percentile FCT for small flows compared to the per-queue ECN with standard threshold.
- **Does MQ-ECN scale to large data center topologies?** Using large-scale ns-2 simulations (§5.2), we show that MQ-ECN scales to multi-hop topologies and delivers the best overall performance. For example, it reduces the 99th percentile FCT for small flows by up to 43.7% compared to the standard threshold, while achieving up to 13.2% lower average FCT for large flows compared to the minimum threshold.
- **How robust is MQ-ECN to network environments and parameter settings?** Using a series of targeted simulations (§5.3), we show that MQ-ECN is robust to 1) the number of queues (§5.3.1), 2) transport protocol (§5.3.2), and 3) parameter settings (§5.3.3).

Benchmark traffic: We use four traffic distributions based on measurements from production DCNs (Figure 5): a web search workload [6], a data mining workload [16], a cache workload [25], and a Hadoop workload [25]. In general, all the workloads are heavy-tailed. Among them, the web search workload and the cache workload are more challenging since they are less skewed. For example, ∼60% of all bytes in the web search workload are from flows smaller than 10MB. Consequently, in the web search workload, it is likely that several flows are concurrently active in the same link, thus increasing network contention. Ideally, different services have different traffic distributions. However, to create more challenges, we hypothetically use the most challenging web search workload for all services in the testbed experiments. We use all the four workloads in the large-scale simulations.

Schemes compared: We evaluate the performance of three schemes, MQ-ECN, per-queue ECN with the standard threshold and per-queue ECN with the minimum



(a) Aggregate goodput (b) RTT

Figure 6: [Testbed] (a) Aggregate goodput of two services achieved by MQ-ECN. (b) RTT distributions.

threshold. We exclude per-port ECN as it can violate weighted fair sharing. For MQ-ECN, there are only two parameters to configure (§3.3.1): β and T_{idle}. In both testbed experiments and simulations, we set β to 0.75 and T_{idle} to the transmission time of a MTU (12μs in 1G testbed and 1.2μs in 10G simulation). We further analyze the sensitivity to above parameters in §5.3.

Performance metric: We use the flow completion time (FCT) as the performance metric. We consider the overall average FCT of all flows and average FCT across different flow sizes (small, medium and large). To evaluate tail latency, we also show the 99th percentile FCT of small flows. For clear comparison, we normalize the final FCT results (not per flow FCT) to the values achieved by per-queue ECN with the standard threshold.

5.1 Testbed Experiments

Testbed setup: We build a small-scale testbed with 9 servers connected to a 9-port server-emulated MQ-ECN-enabled switch. Each server is a Dell PowerEdge R320 with a 4-core Intel E5-1410 2.8GHz CPU, 8G memory, a 500GB hard disk, and the server-emulated switch has 10 Broadcom BCM5719 NetXtreme Gigabit Ethernet NICs. We reserve one NIC on the server-emulated switch for control access. All the servers run Linux kernel 3.18.11 and DCTCP is enabled. We set TCP RTOmin to 10ms as many proposals suggest [6, 19, 30]. On the server-emulated switch, we deploy a MQ-ECN `qdisc` kernel module with 4 queues (per port) scheduled by DWRR. The quantum of each queue is 1MTU. We disable offloading techniques on the switch to avoid large segments. Each switch port has 96KB buffer which is completely shared by all the queues in a first-in-first-serve bias. The base RTT is $\sim$250μs. Given that, we set the standard ECN marking threshold to 32KB.

5.1.1 Static Flow Experiment

We begin with a basic static flow experiment to show that MQ-ECN can achieve high throughput, low latency and weighted fair sharing simultaneously. We start 5 TCP flows from two senders to the same receiver and classify them into two services based on their senders. Service 1 has 1 flow and service 2 has 4 flows. Both services have a equal-quantum dedicated queue on the switch. We evaluate the performance of MQ-ECN and per-queue ECN with the standard threshold.

Figure 6(a) shows the sharing results achieved by MQ-ECN. The sharing result achieved by the standard threshold is quite similar to Figure 6(a). We omit it due to space limitation. In contrast to Figure 3(a), both services roughly achieve the same goodput. We also use ns-2 simulation to reproduce the experiment and find that MQ-ECN achieves similar convergence time as the standard threshold. This suggests that MQ-ECN can strictly preserve weighted fair sharing. Furthermore, the sum of aggregate goodputs of two services achieved by MQ-ECN is $\sim$936Mbps. This suggests that MQ-ECN can fully utilize the link capacity.

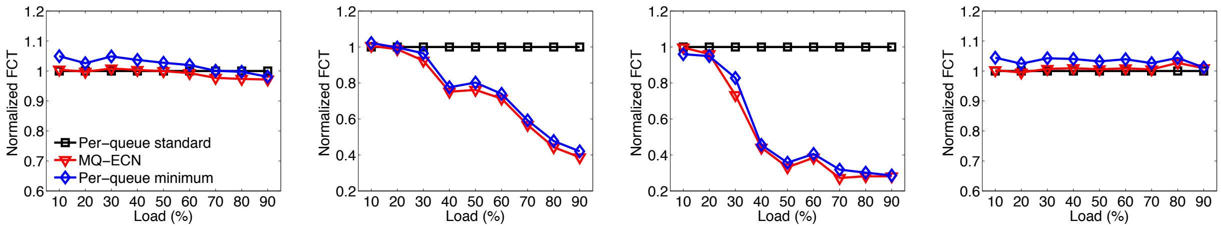
We also measure RTT of the dedicated queue of service 2 using `ping`. Figure 6(b) gives the RTT distributions achieved by MQ-ECN and the standard threshold. Compared to the standard threshold, MQ-ECN achieves 32.3% (651μs to 441μs) and 31.5% (782μs to 536μs) lower RTT in average and the 99th percentile. Recall that the base RTT is $\sim$250μs. Hence, MQ-ECN reduces queueing delay by $\sim$50%. This suggests that MQ-ECN can achieve low latency.

5.1.2 Realistic Workloads

For this experiment, we develop a client/server application to generate dynamic traffic according to the web search workload [6]. The client application, running on 1 server, generates requests through persistent TCP connections to the other 8 servers to fetch based on a Poisson process. The server applications, running on the other 8 servers, respond with requested data. To map a flow to a service queue, the server application uses `setsockopt` to set DSCP for outgoing packets. We create two traffic patterns: *balanced traffic* and *unbalanced traffic*. In balanced traffic, each flow is randomly mapped to a service queue. In unbalanced traffic, each flow is mapped to 4 service queues with probabilities of 10%, 20%, 30% and 40%. We vary the network load from 10% to 90%.

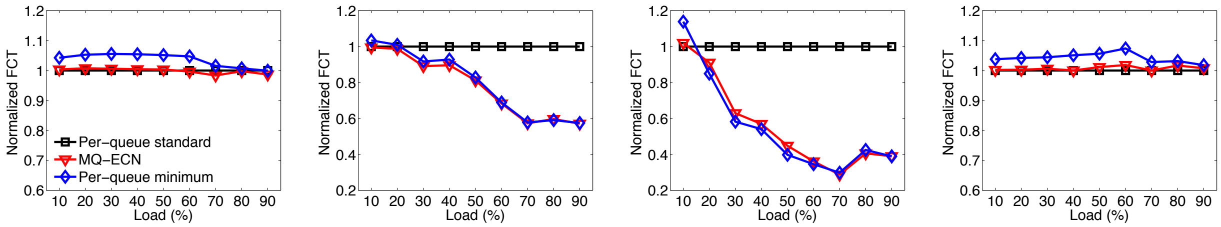
Figure 7 and 8 show the overall average FCT (a), FCT across small (0,100KB] (b,c) and large (10MB,∞) flows, respectively. Due to space limitation, we omit the results for the medium (100KB,10MB] flows whose performance is quite similar to that of overall average FCT. We make the following three observations.

Overall: MQ-ECN generally achieves the best overall average FCT. Compared to the standard threshold, MQ-ECN delivers similar performance at low loads ($\leq$ 50%) and achieves up to $\sim$2.85% (balanced) and $\sim$1.65% (unbalanced) lower FCT at high loads. When the load is



(a) Overall (b) (0,100KB]: Avg (c) (0,100KB]: 99th percentile (d) (10MB,∞): Avg

Figure 7: [Testbed] Balanced traffic pattern: FCT statistics across different flow sizes.



(a) Overall (b) (0,100KB]: Avg (c) (0,100KB]: 99th percentile (d) (10MB,∞): Avg

Figure 8: [Testbed] Unbalanced traffic pattern: FCT statistics across different flow sizes.

low, it is not likely that multiple queues are concurrently active. Hence, both MQ-ECN and the standard threshold can achieve good FCT due to their high throughput. When the load is high, MQ-ECN achieves better performance by efficiently reducing packet latency. Compared to the minimum threshold, MQ-ECN outperforms it at all loads for both traffic patterns.

Small flows: MQ-ECN performs similar as the minimum threshold for small flows while significantly outperforming the standard threshold. Compared to the standard threshold, MQ-ECN reduces the average FCT for small flows by up to ∼61.3% (balanced) and ∼42.9% (unbalanced). The performance gap on 99th percentile FCT is even larger: MQ-ECN achieves up to ∼72.8% (balanced) and ∼71.3% (unbalanced) lower 99th percentile FCT for small flows. We attribute the large tail FCT of the standard threshold to its poor burst tolerance. When all the 4 queues are concurrently active, the total buffer occupancy achieved by the standard threshold can easily reach 128KB (4×32KB), thus overfilling shallow switch buffer (96KB).

Large flows: MQ-ECN also achieves good performance for large flows. Compared to the standard threshold, the average FCT for large flows of MQ-ECN is within ∼2.7% for the balanced traffic pattern and ∼1.8% for the unbalanced traffic pattern. This is expected because MQ-ECN adjusts the ECN marking threshold for each queue based on its dynamic weighted fair share rate, thus not adversely affecting its throughput. By contrast, the minimum threshold, due to its throttle on rates, delivers the worst performance for large flows: it achieves ∼1.2–4.4% (balanced) and ∼1.8–7.4% (unbalanced) larger FCT compared to the standard threshold.

5.2 Large-scale NS-2 Simulations

In this section, we use ns-2 [5] simulations to evaluate MQ-ECN's performance in large-scale DCNs.

Topology: We use a 144-host leaf-spine topology with 12 leaf (ToR) switches and 12 spine (Core) switches. Each leaf switch has 12 10Gbps downlinks to hosts and 12 10Gbps uplinks to spines, forming a non-blocking network. The base RTT across the spine (4 hops) is 85.2μs. We employ ECMP for load balancing.

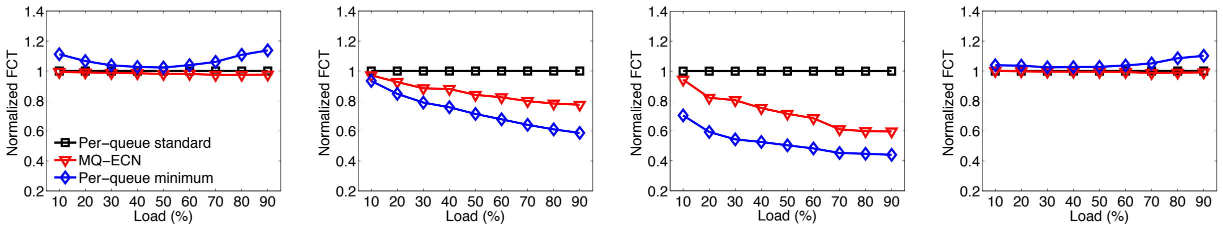
Workloads: We use all the 4 flow size distributions in Figure 5. Since there are 144 hosts, we have 144×143 communication pair in total. We evenly map these pairs to 8 services. Every two services share a flow size distribution. All simulations last for 50000 flows.

Transport: We use DCTCP by default. The initial window is 16 packets. We set both initial and minimum value of TCP RTO to 5ms.

Switch: Each switch port has 300KB buffer shared by all the 8 queues in a first-in-first-serve bias. We set the standard marking threshold to 65 packets. We use both DWRR and WRR in our simulations. All the queues have the same quantum of 1.5KB.

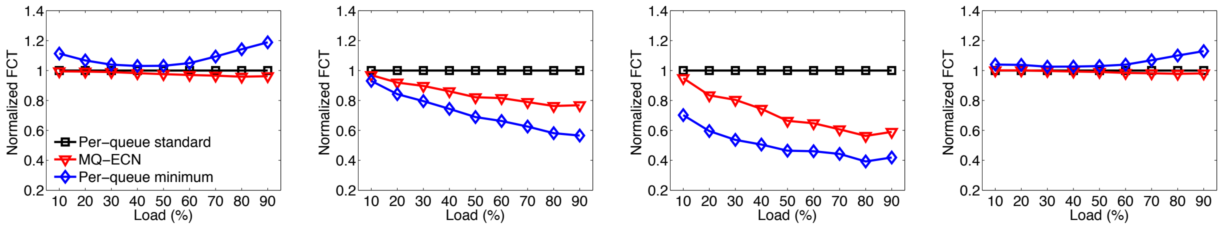
Figure 9 and 10 give the FCT results across different flow sizes. In the interest of space, we omit the results for the medium flows (100KB,10MB] whose performance trend is very similar to that of overall average FCT. We have the following three observations.

Overall: MQ-ECN generally achieves the best overall performance, consistent with our testbed experiments in §5.1. Compared to the standard threshold, MQ-ECN achieves up to ∼4.1% lower average FCT. The performance of the minimum threshold is volatile. Compared



(a) Overall (b) (0,100KB]: Avg (c) (0,100KB]: 99th percentile (d) (10MB,∞): Avg

Figure 9: [Simulation] DWRR: FCT statistics across different flow sizes



(a) Overall (b) (0,100KB]: Avg (c) (0,100KB]: 99th percentile (d) (10MB,∞): Avg

Figure 10: [Simulation] WRR: FCT statistics across different flow sizes

to the other schemes, it shows obvious performance gap at low loads. For example, it achieves ∼11% larger FCT at 10% load compared to the standard threshold. This is because, at low loads, it is likely that only few queues are active. In such scenario, the minimum threshold severely degrades throughput. When the load increases, the minimum threshold shows better performance as expected. However, at extremely high loads, the performance of the minimum threshold degrades again, which is counter-intuitive. We suspect the reason is because ECMP is agonistic to both flow sizes and service classes, and it does not guarantee to spread large flows from the same service class across different paths. When the load is unbalanced or large flows from the same class are concentrated (even overall traffic is balanced), the problem may arise.

Small flows: MQ-ECN greatly outperforms the standard threshold for small flows. Compared to the standard threshold, MQ-ECN reduces the average and 99th percentile FCT for small flows by up to 23.7% and 43.7%, respectively. Compared to the minimum threshold, the average FCT for small flows with MQ-ECN is within 24.3% for DWRR and 26.4% for WRR. The performance gap is because that the minimal threshold trades throughput for better latency. In our simulation, 65.1% of small flows are smaller than 24KB (16 packets), which are small enough to complete within one RTT. The minimum threshold can provide ideal performance for such mice flows since their FCTs are only determined by latency.

Large flows: For large flows, MQ-ECN achieves comparable performance as the standard threshold while significantly outperforming the minimum threshold. This suggests MQ-ECN achieves high throughput. MQ-ECN even slightly outperforms the standard threshold at extremely high loads. For example, compared to the standard threshold, MQ-ECN with WRR achieves 2.1% lower average FCT for large flows at 80% load. This is because MQ-ECN can provide better burst tolerance, thus greatly reducing packets drops and retransmissions. As we check, at 80% load, the standard threshold with WRR causes 720 TCP timeouts while MQ-ECN only has 45. As expected, the minimum threshold performs the worst. For example, it is 13.2% worse than MQ-ECN with WRR at 90% load for large flows.

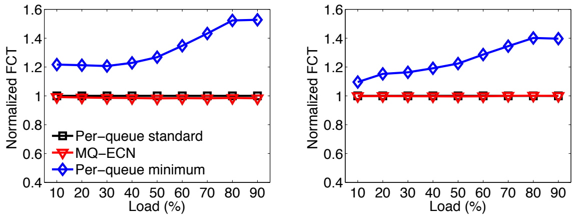
5.3 MQ-ECN deep dive

In this section, we conduct a series of targeted simulations to evaluate MQ-ECN's robustness to network environments and parameters. By default, we use DCTCP as the transport protocol and DWRR (8 queues) as the packet scheduler. The other settings are same as §5.2

5.3.1 Impact of the Number of Queues

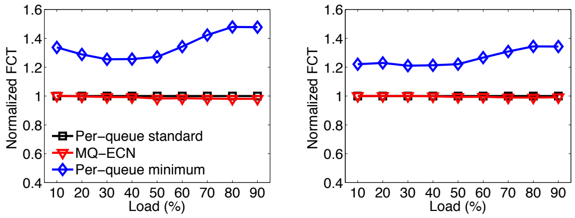
In the future, switching chips may support more and more queues. To verify whether MQ-ECN can scale to a larger number of queues, we increase the number of queues per switch port to 32. In the interest of space, we only show overall performance and average FCT for large flows in Figure 11.

We find that MQ-ECN still maintains the best overall performance. However, the performance of the minimum threshold degrades significantly, particularly at high loads. It is 35.7% worse than MQ-ECN at 90% load for overall average FCT. The reason behind this is: at a given load, the more queues we use, the less likely that the majority of queues are concurrently active. Thus,



(a) Overall (b) (10MB,∞): Avg

Figure 11: [Simulation] FCT with 32 queues.



(a) Overall (b) (10MB,∞): Avg

Figure 12: [Simulation] FCT with ECN*.

the throughput of the minimum threshold is affected, enlarging FCT for large flows. By contrast, MQ-ECN can effectively adjust ECN marking thresholds based on traffic dynamics to maintain high throughput. This indicates that MQ-ECN is robust to the number of queues.

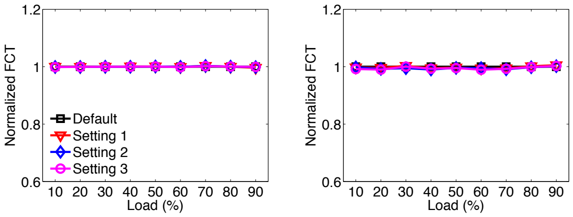
5.3.2 Impact of Transport Protocol

In addition to DCTCP, there are many other ECN-based DCN transport protocols, such as ECN* [31]. Unlike DCTCP, ECN* simply reduces the window by half in the presence of ECN. Hence, ECN* is more sensitive than DCTCP. A lower ECN marking threshold can greatly affect the throughput of ECN* [31]. For example, with zero buffering, DCTCP can maintain 94% throughput in theory [7] while ECN* only achieves 75% throughput.

We evaluate the performance of all the three schemes with ECN*. We set the standard ECN marking threshold to 84 packets. As Figure 12 shows, MQ-ECN still outperforms the other schemes under ECN*. This indicates that MQ-ECN can efficiently maintain high throughput by adjusting ECN marking thresholds based on dynamic weighted fair share rates. As expected, the throughput of the minimum threshold degrades severely. Compared to MQ-ECN, it increases FCT for large flows by ∼22–36%.

5.3.3 Sensitivity to Parameters

We now try to explore MQ-ECN's sensitivity to parameters. Recall that MQ-ECN has two parameters to configure: β and T_{idle}. In our simulation, β is 0.75 and T_{idle} is 1.2μs (1.5KB/10Gbps) by default. Here, we compare the default setting with the other 3 settings: 1) β=0.875, T_{idle}=7.2μs, 2) β=0.5, T_{idle}=1.2μs, and 3) β=0.75, T_{idle}=7.2μs.



(a) Overall (b) (0,100KB]: Avg

Figure 13: [Simulation] FCT with different parameters. FCT is normalized to the value achieved by default setting.

Figure 13 gives the FCT results achieved by above 4 settings. Note that FCT is normalized to the value achieved by the default setting. In general, compared to the default setting, the other 3 settings achieve very close performance. Their performance is within 0.37% for overall average FCT and 1.05% for average FCT of small flows. The results suggest that MQ-ECN is robust to different parameter settings.

6 Related Work

Tons of literatures working along the general ECN/RED, *e.g.*, [6, 12, 15, 24, 29, 31, 32], are related to MQ-ECN. For space limitation, we do not introduce these literatures one by one. However, the key difference is that MQ-ECN is perhaps the first effort that investigates the problem of applying ECN in multi-service multi-queues production DCNs. MQ-ECN does not challenge the fundamental principle of prior work on ECN; instead it builds on the theory (*e.g.*, the standard ECN marking threshold) developed by prior work especially [6, 7, 31], and extends its applicability to a new production environment.

7 Conclusion

In this paper, we have presented MQ-ECN for multi-service multi-queue DCN that is capable of delivering both high throughput and low latency simultaneously, while maintaining weighted fair sharing. We have shown that MQ-ECN achieves all its properties without requiring advanced features and is readily implementable with existing commodity chips. At last, we performed a series of testbed experiments and large-scale simulations to validate its performance as well as robustness to different network environments and parameter settings.

Acknowledgements

This work is supported in part by the Hong Kong RGC ECS-26200014, GRF-16203715, GRF-613113, CRF-C703615G and the China 973 Program under Grant No.2014CB340303. We would like to thank the anonymous NSDI reviewers and our shepherd Changhoon Kim for their constructive feedback and suggestions.

References

[1] http://www.pica8.com/documents/pica8-datasheet-64x10gbe-p3922-p3930.pdf.

[2] "Broadcom BCM56850," https://www.broadcom.com/collateral/pb/56850-PB03-R.pdf.

[3] "DCTCP in Linux kernel 3.18 ," http://kernelnewbies.org/Linux_3.18.

[4] "DCTCP in Windows Server 2012 ," http://technet.microsoft.com/en-us/library/hh997028.aspx.

[5] "The Network Simulator NS-2," http://www.isi.edu/nsnam/ns/.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *SIGCOMM 2010*.

[7] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of DCTCP: stability, convergence, and fairness," in *SIGMETRICS 2011*.

[8] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: trading a little bandwidth for ultra-low latency in the data center," in *NSDI 2012*.

[9] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *SIGCOMM 2013*.

[10] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-Agnostic Flow Scheduling for Commodity Data Centers," in *NSDI 2015*.

[11] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *IMC 2010*.

[12] L. Chen, S. Hu, K. Chen, H. Wu, and D. Tsang, "Towards Minimal-Delay Deadline-Driven Data Center TCP," in *HotNets 2013*.

[13] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, pp. 107–113, 2008.

[14] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4. ACM, 1989, pp. 1–12.

[15] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Netw.*, pp. 397–413.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *SIGCOMM 2009*.

[17] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis," in *SIGCOMM 2015*.

[18] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable Message Latency in the Cloud," in *SIGCOMM 2015*.

[19] G. Judd, "Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter," in *NSDI 2015*.

[20] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. Liu, and F. Dogar, "Friends, not Foes - Synthesizing Existing Transport Strategies for Data Center Networks," in *SIGCOMM 2014*.

[21] J. Nagle, "On Packet Switches with Infinite Storage," *IEEE Transactions on Communications*, vol. 35, no. 4, pp. 435–438, Apr 1987.

[22] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, "Approximate fairness through differential dropping," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 2, pp. 23–39, 2003.

[23] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Transactions on Networking (ToN)*, vol. 1, no. 3, pp. 344–357, 1993.

[24] K. Ramakrishnan, S. Floyd, D. Black *et al.*, "RFC 3168: The addition of explicit congestion notification (ECN) to IP," 2001.

[25] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," in *SIGCOMM 2015*.

[26] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *Networking, IEEE/ACM Transactions on*, vol. 4, no. 3, pp. 375–385, 1996.

[27] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Googles Datacenter Network," in *SIGCOMM 2015*.

[28] I. Stoica, S. Shenker, and H. Zhang, "Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high-speed networks," *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 1, pp. 33–46, 2003.

[29] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," in *SIGCOMM 2012*.

[30] V. Vasudevan *et al.*, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *SIGCOMM 2009*.

[31] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang, "Tuning ECN for data center networks," in *CoNEXT 2012*.

[32] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion Control for Large-Scale RDMA Deployments," in *SIGCOMM 2015*.

DFC: Accelerating String Pattern Matching for Network Applications

Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, Dongsu Han

KAIST

Abstract

Middlebox services that inspect packet payloads have become commonplace. Today, anyone can sign up for cloud-based Web application firewall with a single click. These services typically look for known patterns that might appear anywhere in the payload. The key challenge is that existing solutions for pattern matching have become a bottleneck because software packet processing technologies have advanced. The popularization of cloud-based services has made the problem even more critical.

This paper presents an efficient multi-pattern string matching algorithm, called DFC. DFC significantly reduces the number of memory accesses and cache misses by using small and cache-friendly data structures and avoids instruction pipeline stalls by minimizing sequential data dependency. Our evaluation shows that DFC improves performance by 2.0 to 3.6 times compared to state-of-the-art on real traffic workload obtained from a commercial network. It also outperforms other algorithms even in the worst case. When applied to middlebox applications, such as network intrusion detection, anti-virus, and Web application firewalls, DFC delivers 57-160% improvement in performance.

1 Introduction

Multi-pattern string matching is a performance-critical task for many middlebox applications that perform deep packet inspection (DPI), such as network intrusion detection systems (IDS) [18, 21], Web application firewalls [13, 17], traffic classification [7], and network censorship/surveillance [9, 10, 22]. These applications commonly apply pattern matching to select flows or packets of interest that are subjected to further extensive processing. The common practice for Web application firewalls and IDS is to initially perform pattern matching on the traffic and apply regular expression matching to a relatively small number of packets that contain the string patterns [18, 21, 39, 66]. Many studies have shown that string pattern matching is one of the primary performance bottlenecks for these systems [25, 39, 48, 66, 67].

The key challenge to multi-pattern string matching is that its performance requirement has increased dramatically (e.g., from multi-Gbps [35] to multi-10s of Gbps [6]), outpacing the performance of existing solutions [39, 66]. The popularization of cloud-based third-party middlebox services [58], such as CloudFlare and Akamai's Web application firewall (WAF), requires that they handle large amounts of traffic efficiently [3, 5, 66].

Multi-pattern matching algorithms for network applications must satisfy three requirements: 1) they must support exact string matching to ensure correctness, while identifying the patterns matched, 2) they must support small and variable size patterns, ranging from a single byte to hundreds of bytes [8, 19], and 3) it must provide efficient online lookup against a stream of data (e.g., network traffic) as opposed to batched processing.

The classic Aho-Corasick (AC) algorithm [24] satisfies these requirements, and therefore is used by many applications, including intrusion detection systems, such as those of Snort [18] and Suricata [21], and Web application firewalls, such as ModSecurity [13] and CloudFlare's WAF [66]. AC constructs a deterministic finite automaton (DFA) based on the pattern set and is known to deliver asymptotically optimal performance. However, the main problem is that it references memory frequently and causes a large number of cache misses, resulting in poor performance. In particular, the size of DFA it constructs increases linearly with the number of states and causes severe performance degradation [53]. Although many efforts have been attempted to reduce the memory footprint, they come with undesirable performance tradeoffs because they often require additional computation and/or memory lookups [20].

This paper presents DFC, a memory-efficient and cache-friendly data structure designed to deliver high performance. Our central tenet is that to obtain high performance we must minimize CPU stalls and maximize instruction level parallelism, while reducing the amount of per-byte operations and memory lookups. However, achieving this while satisfying the three requirements is not trivial. In particular, it is especially difficult to support exact matching with *short* and *variable* size patterns. Furthermore, its worst case performance must also be better than that of other algorithms.

To achieve our goal, DFC combines a number of small, efficient data structures by leveraging two key ideas. First, at its heart is a small data structure, called direct filter (DF) that, using a small sliding window, is designed to quickly reject parts of text that will not generate a match. It increases instruction-level parallelism by avoiding data

	1-4 byte	5-8 byte	>9 byte	Total
ET-PRO May 2015	7.3K (28%)	6.7K (26%)	12.1K (46%)	26.2K
ET-Open May 2015	5.3K (30%)	4.6K (26%)	7.8K (44%)	17.7K
Snort VRT 2.9.7.0	1.2K (21%)	1.0K (17%)	3.6K (62%)	5.9K
ModSecurity 2.2.9	0.7K (13%)	1.7K (32%)	2.9K (55%)	5.2K

Table 1: String length distribution in the pattern set

dependencies in the critical path and occupies a small memory footprint (8 KB) for cache efficiency. Moreover, its lookup does not require hash computation. Second, to support exact matching, we use multiple layers of direct filters designed to classify patterns based on their lengths and to filter out the window in a progressive fashion. The key premise is that the longer the pattern is the more one has to inspect to reduce false positives.

Our evaluation shows that DFC outperforms existing solutions by a large margin. In particular, DFC delivers 2.0-2.5 times the performance of AC with 1-26K variable size patterns on real traffic workloads. DFC references memory 1.8 times less frequently and incurs 3.8 times fewer L1 data cache misses. Its memory footprint is four times smaller than that of AC with 26K patterns. Even under malicious and adversarial workloads, DFC outperforms AC by a factor of 1.7 to 2.0. Finally, we implement four applications that use DFC—IDS, Web application firewalls, traffic classification, and anti-virus—and demonstrate that DFC improves their performance by 57 to 160%.

In summary, we make the following contributions:

1. **New algorithm:** We present a new algorithm for exact multi-pattern string matching that gracefully handles short and variable size patterns, and also works well with a stream of network traffic.
2. **Prototype and evaluation:** We evaluate DFC using a variety of deep packet inspection (DPI) patterns from IDS, anti-virus software, and Web application firewalls under real and synthetic workloads.
3. **Real-world applications:** We show four applications of DFC. When applied to intrusion detection, DFC delivers up to 2.3X the performance of the next best solution, in real traffic from a commercial ISP. When applied to Web application firewalls, DFC delivers 1.9X the performance. Moreover, DFC improves the performance of traffic classification and anti-virus by 60% and 75%, respectively.

2 Motivation

Why is it important? Network applications, such as network intrusion detection, traffic classification, and Web application firewalls, specify pattern signatures using regular expression and strings [48, 53]. While regular expression is much more *expressive*, matching multiple regular expressions [60] is much more *expensive*. Matching a *single* regular expression (regex) against network traffic is 4X slower than multi-string matching with 2.4K patterns [39][§4.5]. Matching multiple regex patterns by constructing a single extended automaton (XFA [60, 61]) is at least two orders of magnitude slower than multi-string matching.[1] Due to the distinct tradeoff between performance and expressiveness of the two, string pattern matching is commonly used to accelerate regex-matching. String matching can filter out the vast majority of input early on and specifies a small set of candidate regex patterns to inspect [13, 18, 21, 31]. Typically, regex patterns contain at least a few bytes of string [60][Table 2]. SplitScreen [31] leverages this to extract strings from regex patterns and uses them as a pre-filter to improve the performance. For the same reason, it is strongly recommended that IDS signatures contain string fields [23].

String matching is also used for protocol classification/identification and for testing the presence of keywords. Because string-pattern matching is a critical building block for network applications that inspect payloads, it is reported that string matching is one of the most expensive operations in Web application firewalls [66] and accounts for 70 to 80% of CPU cycles for IDS [25, 39].

Today, middlebox services offered over the cloud (e.g., CloudFlare's WAF [66]) allows customers to easily sign up for these services. One of the reasons for the widespread use of WAF is that it is one of the two ways to meet Payment Card Industry (PCI) Data Security Standards (DSS) requirement 6.6 for Web sites that take credit cards [1, 16]. [2] Decrypted traffic is inspected by Note, a Web application firewall (WAF), to protect the Web service. For this, some WAFs [13] run as part of the Web server, whereas cloud-based WAF services integrate itself with existing SSL acceleration service. We believe that popularization of cloud-based middlebox services and the security regulations/best practices make high-performance multi-pattern matching more important.

Workload: Three key characteristics define our target workload. First, patterns can appear *anywhere* in the text. Second, the number of patterns is large, typically on the order of 10K patterns, and accumulates over time as new patterns are discovered. Finally, patterns are typically *short* and of *variable* size.

Table 1 shows the pattern-length distribution from four popular rulesets for IDS and Web firewalls. Over 54% of patterns are 8 bytes or shorter in the ET-Pro®ruleset, a popular commercial pattern set for IDSs [8].

Why is it difficult? We categorize existing approaches into three classes and show that the characteristics of our workload make each of these algorithms rather ineffective.

[1] Note, the main benefit of XFA is that it solves the state explosion problem of DFAs. But its matching performance is still lower than that of DFA. Although it is an indirect comparison, XFA's multi-regex matching takes hundreds of CPU cycles per byte of input [60, 61], whereas multi-string matching takes a few cycles.

[2] The other option is to conduct application vulnerability security reviews of all Web applications in use.

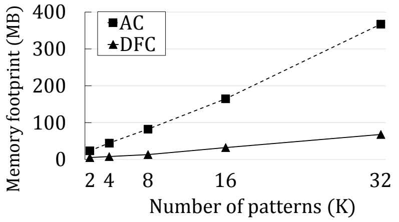


Figure 1: Memory footprint of AC and DFC

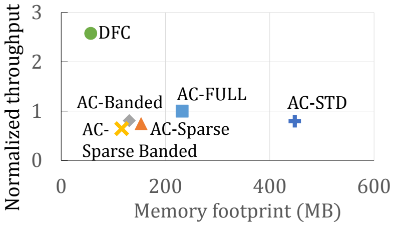


Figure 2: Throughput and memory footprint

	DF	DFC	AC
# L1-D cache load	2.0	4.6	8.4
# L1-D cache misses	0.003	0.282	1.070
# L2 cache misses	0.002	0.036	0.190
# L3 cache misses	0.0019	0.011	0.017

Table 2: Number of memory accesses and cache misses per byte of input text

Aho-Corasick-based algorithms construct finite state automata (based on the pattern strings) that consume each character of the input text sequentially. AC is by far the most popular multi-string matching algorithm used for deep packet inspection. While it achieves asymptotically optimal performance, in practice, its performance is far from ideal because it generates frequent cache misses due to its large memory footprint [53]. As shown in Figure 1, its memory footprint increases dramatically as the number of patterns increases. [3] For a reasonably large pattern set, the data structure does not fit in a CPU cache, resulting in a severe performance penalty.

Variants of AC [20, 55] that try to compress the state transition tables come at the cost of decreased performance. Figure 2 illustrates the memory footprint and performance trade-offs of five commonly used AC variants in Snort [20, 55], in contrast with those of DFC. For AC implementations, the memory footprint and the throughput are highly correlated. As a result, Snort defaults to AC-FULL [20], which achieves the highest throughput at the cost of the largest memory footprint.

The AC algorithm also frequently accesses memory because it examines the state table for every byte of text. Table 2 shows the average number of data accesses (measured using # of L1 data cache accesses) and the number of L1 data cache misses per byte of input text, while matching randomly generated text with 26K patterns from the ET-Pro ruleset.[4] AC accesses the memory at least five times on average per byte of input text, consisting of access to: (1) the state machine, (2) the input text, (3) the case translation table (for case insensitive lookup), (4) the state transition table for looking up the next state, and (5) a field that indicates whether the next state indicates a valid match. Note, DFC reduces the memory access frequency by a factor of 1.8 as shown in §5.4.

Heuristic-based algorithms try to achieve sub-linear time complexity by advancing the sliding window by multiple characters using the "bad character" and "good suffix" heuristics [30]. For example, the Wu-Manber algorithm [71] leverages the "bad character" heuristics. After examining a block of characters, it looks up a shift table that indicates by how many bytes it can shift the sliding window. The shift table is constructed so that when a block of characters never appear in any of the patterns, the algorithm skips the entire block and advances the sliding window by multiple characters. However, this has two main limitations. First, heuristics works well when the pattern is long and its size is fixed, but is not effective with *short patterns* [48, 71]. With short patterns, the shift distance becomes small because the block size has to be less than or equal to the shortest pattern, which is one byte in our workload. Second, the algorithm inherently has data dependencies that make it difficult to leverage the performance characteristics of modern CPUs. Until it retrieves the shift value from memory, it cannot determine the next window to examine. This limits the instruction-level parallelism, results in frequent instruction pipeline stalls, and makes prefetching very difficult. As a result, heuristics like Wu-Manber introduce severe performance penalty in practice, as we show in §5.4. [5]

Hashing-based algorithms compare the hash of a text block with the hash of the pattern. They are designed to quickly filter out non-matching text using its hash values, but have several practical limitations. First, they introduce false positives and thus require additional processing to ensure exact pattern matching [53]. Second, when the patterns are of variable size, the text block to hash must be shorter than or equal to the *shortest* pattern to avoid false negatives, which makes the algorithm ineffective under our workload. A common solution is to use hashing for long patterns and to fall back to traditional approaches (e.g., Aho-Corasick) for short patterns. For example, an exact matching algorithm based on feed-forward Bloom filters (FFBF) [53] applies the hash-based approach for patterns of size greater than 19 bytes, but uses AC for the remaining patterns. Finally, they require multiple expensive hash computation for *every sliding window*. Bloom filters typically use multiple hash functions applied on every sliding window. Even if a rolling hash function [53] is to incrementally calculate the hash values, it is far more expensive (than a simple lookup) as we show in §5.4.

[3] The patterns were taken from Snort and ET-Pro rulesets [8, 18].

[4] The performance statistics are obtained using *perf* and Intel Performance Counter Monitor [12].

[5] Note earlier versions of Snort once used a modified Wu-Manber algorithm.

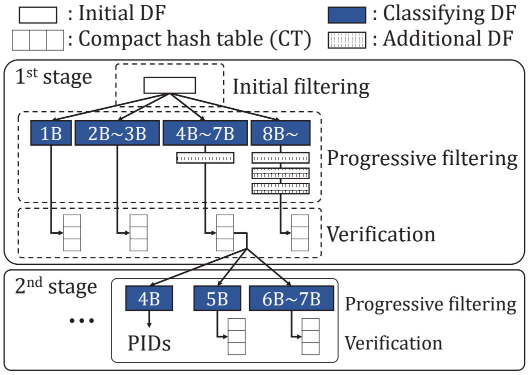


Figure 3: DFC design overview with an example configuation

3 Design

To overcome the limitations of existing approaches, our main approach is to develop an efficient and simple primitive that we use as a key building block. Making the basic component efficient is critical because the nature of the workload—matching a large number of small and variable sized patterns—forces us to examine *every byte* of input text. We show that one can design an efficient string matching algorithm out of a simple primitive. We also show that our algorithm not only delivers the best average case performance but also is robust across many workloads, including the worst case.

Figure 3 shows the overall design of DFC that consists of three logical parts:

- **Initial filtering** phase employs a simple filter that eliminates windows of input text that fail to match any string patterns. Only the input text window that is not filtered advances to the next phase. We describe the filter design in §3.1.
- **Progressive filtering** phase first determines the approximate lengths of potentially matching patterns. We group patterns into pattern size groups (e.g., 1B, 2-3B, 4-7B, 8+B) and use *classifying filters* to determine the groups that the current window belongs to. We then apply additional filtering proportional to their lengths using *additional filters*. We use the same type of filters for both purposes. We describe the design of progressive filtering in §3.2.
- **Verification** phase verifies whether the input generates an exact match by comparing the text with actual patterns. This is required because the previous phases do not completely eliminate false positives. For example, if there are patterns 'AABB' and 'CCDD', the initial DF can be set for 'AA' and 'CC' and an additional DF can be set for 'BB' and 'DD'. In this case, an input sequence 'AADD' will pass both filters even though it is not a match. For verification, we use a compact hash tables for each pattern size class. Depending on the pattern size group determined by the previous phase, it inspects a different table. Its lookup is efficient because the progressive filtering phase significantly reduces both the false positives and the set of potentially matching patterns to inspect. We describe the design in §3.3.

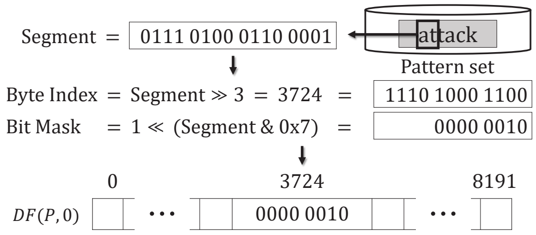

Segment = 0111 0100 0110 0001
attack
Pattern set
Byte Index = Segment ≫ 3 = 3724 =
1110 1000 1100
Bit Mask = 1 ≪ (Segment & 0x7) =
0000 0010
0
3724
8191
DF(P, 0)
...
0000 0010
...

Figure 4: An initialization example of Direct Filter

Two-stage hierarchical design: Our algorithm uses progressive filtering and verification in two stages as shown in Figure 3. The first stage of progressive filtering classifies patterns by their lengths in a coarse grained manner. After roughly determining the pattern lengths, it looks up a hash table for the pattern class in the verification phase. However, this may be inefficient if there are many hash collisions. Some buckets may contain many patterns due to the skewed popularity of patterns. Normally, hash collisions can be controlled either by using a more uniform hash function or adjusting the size of hash table. However, we find that hash collisions are caused because real-world pattern sets often contain common string prefix or segments, and some prefixes (e.g., HOST) appear very frequently (e.g., more than 100 times). For example, consider a hash table indexed by 4 bytes and holds patterns of size 4–7B. If many of them start with the same prefix (e.g., HOST), then all of them will go to the same bucket. This dramatically worsens the worst case performance because we must perform verification. However, in this case, increasing the hash table size or using a better hash function does not help, but only increases the overhead.

To remedy this, we selectively employ 2nd-stage progressive filtering on a per-bucket basis. When a bucket contains a large number of patterns, we apply a 2nd stage of progressive filtering for finer-grained classification as shown in Figure 3. For buckets with many collisions, the 1st stage verifies exact matching with the prefix (minimum pattern length in the group) and the 2nd stage only examines the string that follows the prefix.

3.1 Filter Design

DFC design relies on an efficient filter primitive, called direct filter (DF), for filtering and classification.

Direct filter is a bitmap indexed by several bytes of input text. For a 2B DF, a 2B sliding window from the input is treated as a 16-bit unsigned integer and used as a bit index to DF. Each bit tells whether the string containing the 2B window can generate a match with any patterns. For most cases, we use 2B windows so that the DF fits in lower levels of the CPU cache. A 2B DF is initialized by taking a

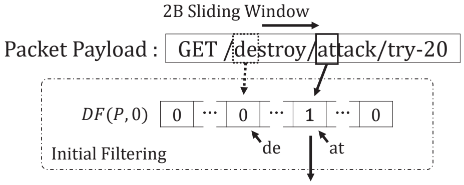


Figure 5: Initial filtering example

single two-byte fragment (e.g., first two bytes) from each string pattern and setting the bit indexed by the two bytes, while the rest of the bits are set to zero. Figure 4 shows an initialization example of DF for a pattern 'attack'.

During the initial filtering phase, we take every two-byte window of the input text and look up whether the corresponding bit is set in the DF as shown in Figure 5. A DF lookup performs three bitwise operations (two SHIFTs and one AND) and a single memory reference. If the corresponding DF bit is not set, it indicates that the part of input text that contains the two-byte window cannot generate a match. Thus, no pattern of interest can appear at this location of the input text. We then advance the window by one byte to examine the next window. If the corresponding bit is set, it indicates that the location of text potentially contains patterns of interest. Only then, we inspect in more detail the sequence starting from the current window.

The initial filtering phase uses a single DF to filter out most of the innocent input text. In our experiment with real traffic and 26K patterns from the ET-Pro ruleset, 94% of windows are filtered out in this phase.

Size of initial DF: Similar to other filters (e.g., Bloom filter [29]), the size of DF determines the tradeoff between the number of cache misses and false positives; DF is actually a special case of Bloom filter that uses a single identity function as the hash function. For most cases, we use a two-byte indexed 8KB DF (2B DF) to achieve a balance. If we use one-byte indexed DF (1B DF), the size of DF reduces to 256 bits, but the rate of false positives will be 256 times higher on average. This would, for example, cause 34.8% of windows with 26K ET-Pro rules (compared to 3.6% for 8 KB DF assuming a uniformly random input) to advance to the next phase. In contrast, using a three-byte indexed DF (3B DF) further reduces the false positives, but the size of DF increases to 2 MB. Because initial DF lookup is performed very frequently, we would like to minimize the cost by making DF fit inside lower levels of the CPU cache (2 MB easily exceeds the typical size of L2 caches).

Our evaluation shows 2B DF delivers better performance for workload that contains up to a few tens of thousands of patterns; using a 3B DF is actually up to 18% slower as we show in §5.4. This is because L3 cache latency is four to seven times higher than that of L2 cache [47]. Thus, we use 2B DF in most cases, but use

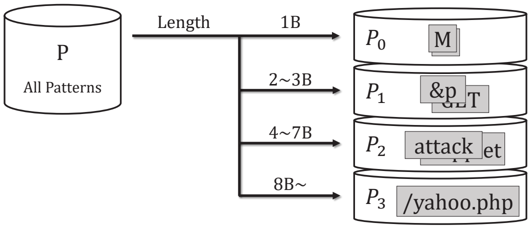


Figure 6: A classification example of patterns by lengths

3B DF when there are millions of patterns (e.g., ClamAV ruleset). We quantify this tradeoff in more detail in §5.4.

Initialization: To make initial filtering more effective, we try to minimize the number of bits set. A negative correlation exists between the number of bits set in the initial DF and its performance because the more the bits are set, the more windows pass through. We observe that minimizing the number of bits set in the initial DF delivers up to 10% performance benefit than a native solution (i.e., using the first two byte) in the real traffic workload. We use the following heuristics to achieve this. First, we classify patterns into their lengths as we do for the progressive filtering phase. We use the same offset for each class to avoid looking up additional state in progressive filtering. In particular, we examine the pattern class size in the increasing order (i.e., from short patterns to long patterns) and, for each class, we choose the offset of the two-byte segment from each class so that it minimizes the number of additional bits set.

If 1-byte patterns exist, we enumerate all two-byte segments that start with the single byte in the pattern. Thus, a 1-byte pattern sets 256 bits in the DF. Each pattern of size two bytes or more sets a single bit if they are case sensitive. If they are case insensitive, we enumerate all cases, setting at most four bits per string pattern.

3.2 Progressive Filtering

The main idea of this phase is to progressively eliminate false positives to take small steps towards exact pattern matching using multiple layers of DFs. The key insight we leverage is that the longer the pattern is, the more one has to compare it against the input text to reduce false positives. For example, when the text is filtered using a single DF, a two-byte pattern does not produce any false positive, but four-byte patterns can produce as many as 2^{16} false positive patterns.

Thus, we first determine the pattern lengths that the current window of input might match and apply different amounts of additional filtering proportional to the lengths For example in Figure 3, we use three additional DFs for 8+ bytes but only use a single additional DF for 4-7 bytes in the 1^{st} stage progressive filtering. As the algorithm progresses, the set of the potentially matching patterns also reduces, and the likelihood of generating a match increases exponentially. When the scope is sufficiently

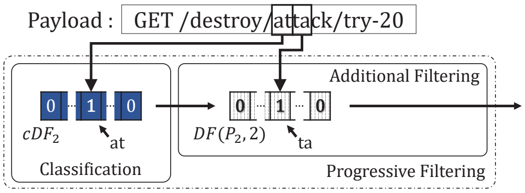


Figure 7: Progressive filtering example

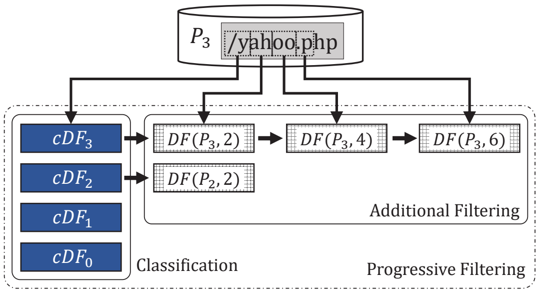


Figure 8: An initialization example of cDFs and DFs in Progressive Filtering

reduced, we perform exact comparison in the verification phase. Progressive filtering consists of two steps: pattern length classification and additional filtering.

Classification: During initialization of DFC, we classify the string pattern set P into pattern classes (P_i's) according to their lengths—P_0 contains the shortest patterns and $P_1, ..., P_n$ contains longer patterns in increasing order. Figure 6 shows an example with four classes (e.g., P_0 contains 1-byte patterns, and P_3 contains patterns of 8 bytes or longer). For each class, P_i, we create a classifying direct filter, cDF_i, whose bits are set using the patterns in P_i. For patterns in P_i, to set the cDF_i, we use the same two-byte segment that was used to set the initial DF.

At run time, we look up the classifying DFs (cDFs). Each cDF determines whether the window might match the patterns in the class. For example if the corresponding bit of cDF_i is set, we know that we must inspect the pattern class P_i. Note the classification is *not* mutually exclusive; multiple pattern classes can match because patterns from two different classes may share the same two-byte segment.

Additional filtering: When the corresponding bit in cDF_i is set, we further filter the input sequence that follows the current two-byte window with additional DFs. These additional DFs are designed to inspect different two-byte text segments in the pattern. We use additional DFs, $DF(P_i, O_k)$'s, to test whether the input text might match patterns in P_i, where O_k denotes the offset from the initial window—e.g., the initial DF that inspects the first two byte can be represented as $DF(P, 0)$. $DF(P_2, O = 2)$ inspects the third and fourth bytes (two bytes at offset two) against patterns in P_2 as depicted in Figure 7; The following two-byte text segment is inspected using additional DF ($DF(P_2, 2)$) after passing through classifying DF (cDF_2). $DF(P_2, O = 2)$ is thus created from using two bytes from P_2 that correspond to offset O. Figure 8 depicts how cDF and additional DFs are initialized for a pattern from the pattern class P_3. Each DF performs filtering and we advance to the next phase only if the window passes through all DFs in the sequence. Algorithm 1 shows the pseudo code for progressive filtering and verification.

Algorithm 1: Progressive Filtering and Verification

```
# @param  cDF   Array of Classifying Direct Filters
# @param  buf   Input text
# @param  pos   Position of window
def Progressive_Filtering(cDF, buf, pos):
    for $i = 0$ to $\#\,of\ cDF$:
        if Corresponding bit of buf at pos in cDF[i] is set:
            if Pass through additional DFs:
                CT ← corresponding hash table
                pos ← corresponding position
                Verification(CT, buf, pos)

# @param  CT    (Compact) Hash Table
# @param  buf   Input text
# @param  pos   Position of window
def Verification(CT, buf, pos):
    H ← hash value of buf at pos
    Bucket ← CT[H]
    Second ← Second stage flag in Bucket
    if Second is true:
        cDF ← array of cDFs in Bucket
        pos ← corresponding position
        Progressive_Filtering(cDF, buf, pos)
    else:
        for PID ∈ Bucket:
            Perform exact matching for PID with buf at pos
```

Additional filtering is an optimization to avoid the verification phase where we perform hash table lookup and exact string comparison. It is only beneficial, if the benefit outweighs the cost of additional DF lookups. In general, the longer the pattern size, we inspect longer segments that follow the current window. However, because each additional lookup adds decreasing marginal benefit, we use a small number of additional filters. We discuss the configuration issue at the end of the subsection. As we show in §5, each additional filtering is effective, and once the input text passes through this phase, it is much more likely to generate a match.

Optimizations: To minimize the average memory lookup, we perform two optimizations. First, similar to the initial DF, we carefully choose the offsets of each additional DFs to filter text as much as possible. For this, we choose an offset that minimizes the number of bits set for each DF. This is done when the DFs are initialized. To reduce the search space, we first identify non-overlapping two-byte segments and greedily select the offset. Second, we change the order of DFs we inspect so that the most effective filter comes early. Note that each sequence of DFs that do not share a parent (e.g., cDF(P_3,0) → DF(P_3,2) and cDF(P_2,0) → DF(P_2,2)) is independent, and the ordering of DFs within a sequence does not affect the correctness of the algorithm. However, if we first look up the one with the smallest number of bits set, the number of average memory lookups can decrease. Thus, we

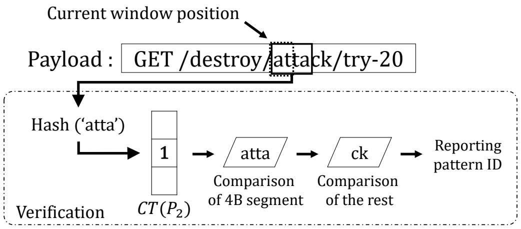


Figure 9: Verification example

order DFs in a non-decreasing order of the number of bits set.

Configuration: The number of pattern classes and their ranges are parameters that can be configured. In our evaluation with DPI patterns in Table 1, we classify the length in multiples of two—P_i holds patterns of length 2^i to $2^{(i+1)} - 1$, and the last class holds the rest of the patterns. Figure 6 illustrates our configuration. P_0 holds patterns of one byte and P_1 holds patterns of two to three bytes. We classify patterns into four classes by their lengths: one-byte (P_0), two to three bytes (P_1), four to seven byte (P_2), and 8+-byte (P_3) string patterns. In our implementation, we use an additional filter for P_2 and three additional DFs for longer patterns (≥ 8).

The classification we use in our implementation is determined empirically and can be tuned depending on the pattern set; e.g., when there is no pattern of size [2,3], having such a classification is wasteful. Tuning the classification parameters involves in one time cost that can be done offline when the pattern size distribution has changed significantly since the last update. On the one hand, having a fine-grained classification or a large number of classes helps in significantly reducing the false positives in the progressive filtering phase and the number of patterns to inspect in the verification phase. This is because we can only filter up to the minimum length in each pattern set to ensure that there is no false negatives. However, fine-grained classification increases overhead because each DF requires an additional memory lookup. Thus, we must strike a balance in choosing the granularity of the pattern classes. We evaluate the tradeoffs of having a finer or coarser classification in §5.

3.3 Verification

The final verification phase ensures exact matching by comparing the text with the actual patterns in the pattern class. To perform exact matching, we create a (compact) hash table, $CT(P_i)$, for each pattern size class, P_i. These tables are indexed by a hash of text fragments whose lengths correspond to the shortest pattern in P_i; e.g., for pattern class P_2 of size 4 to 7, a hash table $CT(P_2)$ is indexed by a hash of four-byte segment. Each bucket in the hash table holds a list of a pattern ID (PID), a text segment of the pattern, and a pointer to the rest of the pattern text.

Rulesets		
Rulesets	ET-Pro (May 2015) Snort VRT 2.9.7.0	ET-Open (May 2015) ModSecurity CRS 2.2.9
Input workload	(1) Random payload (2)Real traffic traces from a commercial ISP	

Table 3: Patterns and inputs

Total volume	68 GB
Number of packets	89,043,284
Number of HTTP packets	77,582,806
Number of TCP sessions	2,213,975
Number of HTTP sessions	1,869,208
Capture duration	57 min 16 sec
Average packet size	757 B
Average flow size	30 KB

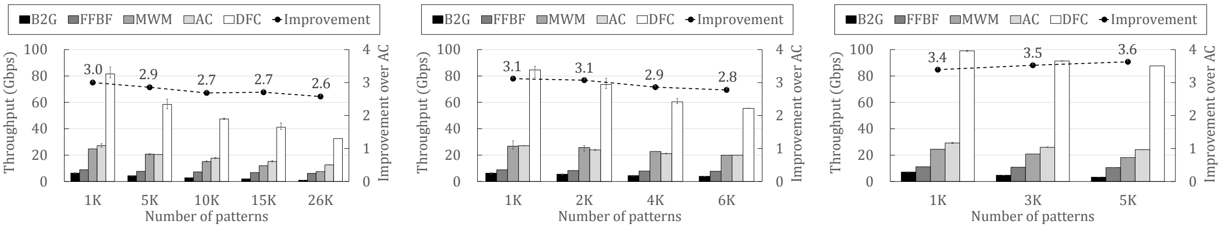
Table 4: Statistics of traffic traces from a commercial ISP

During verification, we compare the text segment with all the pattern segments in the bucket. If a match is found, we compare with the rest of the string segments to ensure an exact match. If exact matches are found, we report the PID and the offset within the input text that generated the matches; Figure 9 illustrates verification process for 'attack' pattern from pattern class P_2. A hash table $CT(P_2)$ is indexed by a hash of four byte segment 'atta' because a length of the shortest pattern in P_2 is four. Then, the four byte segment is compared to four byte segment in the bucket. Because they are same, the rest of the pattern 'ck' is compared to the following two-byte segment from the payload and PID of the pattern is reported. On the one hand, using a simple hash function is good enough because the number of possible patterns that can actually reach this phase is significantly reduced due to progressive filtering.

We adjust the size of the table so that on average a small number of PIDs are present (e.g., < 0.1 PIDs). However, because pattern strings commonly share some popular string segment hash collisions may be high for some buckets as explained in §3. If the number of PIDs in a bucket exceeds a threshold, we apply the second stage on a per-bucket basis as represented in Algorithm 1. In this case, the 1st stage examines up to the minimum length pattern in its size classification, and the 2nd stage verifies the rest. In the 2nd stage progressive filtering, we perform a more fine-grained classification and verify the rest of the text in the second stage of verification (Figure 3). For example, for P_2 that holds patterns of size [4,7], the 2nd stage progressive filtering divides it into three sub-classes. The 1st stage examines up to four bytes and the 2nd stage examines the rest.

4 Implementation

DFC is implemented in 2.4K lines of C code. For comparison, we use AC and modified Wu-Manber (MWM) algorithm implementations in Snort, b2g algorithm (2-gram implementation of the SBNDMq) [36] implementation

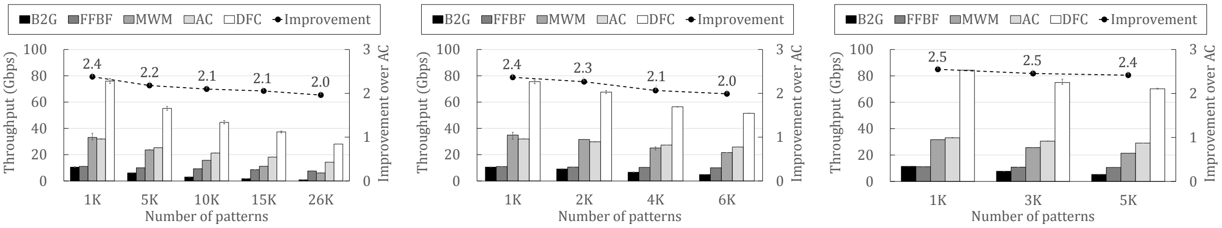


(a) ET-Pro rulset (May 2015)

(b) Snort VRT 2.9.7.0 ruleset

(c) ModSecurity 2.2.9 ruleset

Figure 10: Standalone performance benchmark of AC and DFC under random packet workload.



(a) ET-Pro rulset (May 2015)

(b) Snort VRT 2.9.7.0 ruleset

(c) ModSecurity 2.2.9 ruleset

Figure 11: Standalone performance benchmark of AC and DFC using packet contents from real traffic.

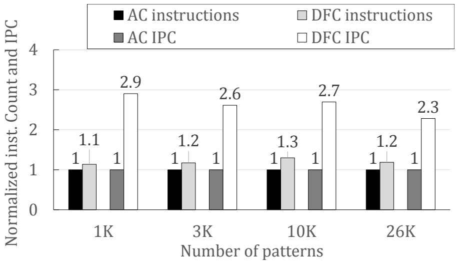


Figure 12: Instruction count and IPC from the benchmark for real traffic with ET-Pro ruleset

from Suricata, and FFBF [53]. In FFBF, short patterns are processed by AC and long patterns are processed by FFBF. FFBF works as a filter and outputs a subset of patterns and input set that can potentially match. Then, it uses AC to remove false positive [53]. To demonstrate the real-world benefit, we apply DFC to four different applications: intrusion detection system, Web application firewall, HTTP traffic classification, and anti-virus. For IDS, we choose an optimized version of Snort, Kargus-CPU, that uses high performance packet I/O and lightweight data structures [39]. Note, we do not use its GPU module, but only use its CPU version. Its pattern matching algorithm is almost identical to that of Snort. We replace AC with DFC by modifying 82 lines of code. For Web application firewalls, we apply DFC to ModSecurity, a popular open source implementation. ModSecurity scans through HTTP requests and responses to match against malicious patterns. For HTTP traffic classification, we use nDPI [7]-like traffic classification for HTTP traffic that looks for popular domain names in the HOST field of the request header. This classifies which application (e.g., Netflix, YouTube) generated the traffic. Note, HTTP consists of 75% of all downstream bytes in modern cellular networks [70]. For these three implementations, we thoroughly conduct correctness tests by comparing the result from DFC with result from AC using two types of input: randomly generated input and a 68GB traffic trace from a commercial cellular ISP. We find that the DFC produces output identical to AC. Finally, for anti-virus, we modify ClamAV [43] version 0.98.7.

5 Evaluation

We answer four questions about DFC in this section:

- How does it compare with existing algorithms under a variety of workloads?
- How much gain does it provide when applied to middlebox applications?
- How does each component affect performance, and what are the trade-offs involved in parameter settings?
- How does its direct filter compare against other primitives?

5.1 DFC Performance Benchmark

We compare the performance of DFC with that of AC, modified Wu-Manber (MWM), FFBF, and b2g across various workloads and configurations. We empirically choose a window size of FFBF to 32B and a size of bloom filter to 1MB and use 4 hash functions, showing the best performance in the workloads and configurations. B2g does not detect one byte patterns because it processes 2 characters as a single character.

To evaluate the performance on a pure algorithmic basis, we disable network I/O and feed input directly from memory. We use four different DPI patterns and two kinds of input workloads described in Table 3. The DPI patterns are taken from popular IDSs and a Web firewall, and the input workloads are either randomly-

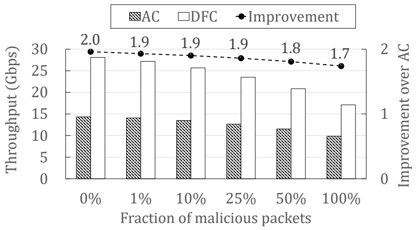


Figure 13: Performance with malicious traffic (ET-Pro 26K rules)

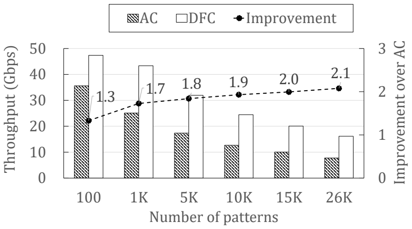


Figure 14: Pathological workload: Input purely consists of all two-byte segments used to set initial DF

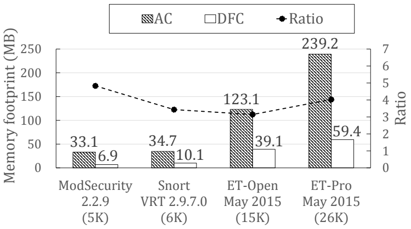


Figure 15: Memory footprint comparison

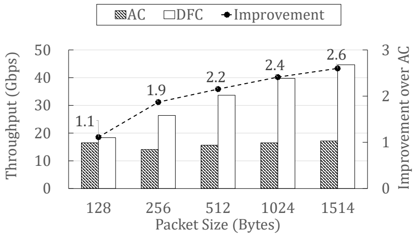


Figure 16: End-to-end IDS performances under synthetic traffic (Snort VRT 6K rules)

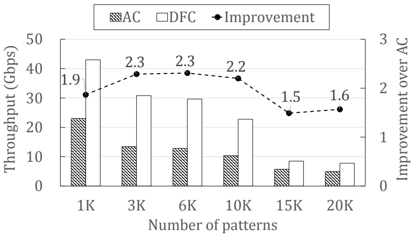


Figure 17: End-to-end IDS performances with traffic traces from a commercial ISP

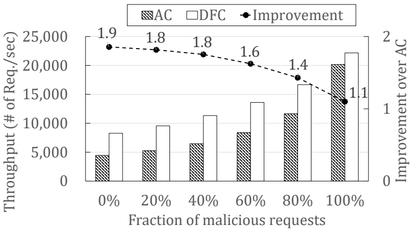


Figure 18: Performance of Apache with ModSecurity with malicious HTTP requests

generated or taken from a real traffic dump from a 10 Gbps LTE backhaul link at a commercial cellular ISP. Table 4 summarizes the statistics of the real traffic trace. Unless specifically noted, the results are measured on an Intel E5-2690 (Sandy Bridge) machine with 16 cores and 126 GB of memory.[6] We use Intel®Compilers (icc) for all experiments. We inspect the input against all patterns in the pattern set. Note some applications, such as Web firewall and anti-virus, match the input against all patterns, while other applications, such as IDS, specify the flow group in 5-tuple with the patterns and only the packets that belong to the group are matched against the patterns.

Figure 10 and 11 show performance comparisons for each input data, while varying the number of randomly sampled patterns. We report the average of ten independent runs. The error bars in figure show min and max of throughput. Dotted line shows DFC's performance improvement over AC. [7]

With the random traffic workload, DFC outperforms AC by a factor of 2.6 to 3.6. B2g performs worse than AC even though it does not detect one byte patterns. FFBF also shows worse performance than AC due to short patterns in the ruleset (Table 1) that requires traditional Aho-Corasick based matching. MWM also performs slightly worse than AC. The existence of short patterns contributes to the loss in performance. With the real traffic workload, DFC outperforms AC by a factor of 2.0 to 2.5. DFC's relative improvement slightly decreases as the number of patterns increases. This is because the text generates more matches. The difference between the real traffic and random payload is also due to the same reason. The real traffic contains a relatively larger fraction of string segments in the pattern set because some string patterns contain very generic keywords (e.g., GET). Note the rules that contain these generic patterns also have other options to evaluate (e.g, traffic direction and user-agent type). In a real IDS setting, these packets will be filtered by such options. Our microbenchmark is a conservative one that purely focuses on the string matching. Nevertheless, DFC consistently delivers significant performance improvement in all cases.

Figure 12 shows the number of instructions executed and the instructions per cycle (IPC) of DFC relative to those of AC. We observe that DFC executes 10 to 30% more instructions, but the instruction throughput improves by 130 to 190%; as shown in Table 2, DFC incurs L1 cache misses 3.8 times less frequently and memory accesses 1.8 times less frequently compared with AC.

Malicious traffic and pathological cases: We now evaluate DFC under malicious input traffic by varying the fraction of packets that contain a pattern string from the ruleset. We insert a randomly selected malicious pattern for each packet both for the real traffic workload and random traffic. Figure 13 shows the result with 26K patterns in the ET-Pro ruleset with real traffic. Both algorithms' performance degrades by approximately 30% as we increase the fraction of malicious packets from 0 to 100%. DFC consistently outperforms AC by a factor of 1.7 to 2.0 because AC's performance also degrades due to increased cache miss rates and the additional cost of book-keeping the position of matched input and the PID. Even when all packets contain a malicious pattern, it achieves 1.7 times the performance of AC. We observe a similar behavior

[6] We use two CPUs with 16 cores. The actual memory usage depends on the algorithm (see Figure 15).

[7] We omit ET-Open result because it is similar to ET-Pro's.

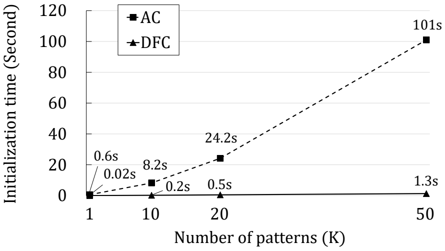


Figure 19: Time to initialize data structures

in the random traffic workload (not shown in the figure) with performance improvement by a factor of 2.2 to 2.6.

To study the behavior of DFC with more adversarial input, we generate a pathological case where the input consists only of two-byte segments used to set the initial DF. We randomly order the two-byte segments to generate the input. This forces DFC to pass through the initial filter on every two bytes of input. Figure 14 shows the performance of AC and DFC by varying the number of patterns. Even in this case, DFC performs significantly better than AC regardless of the number of patterns.

Finally, we examine the two cases where *every byte* of input is malicious and nearly malicious. For the first case, we construct the payload by concatenating randomly selected the strings from the pattern set. For the second case, we concatenate all but the last byte of randomly selected patterns, forcing DFC to trigger the verification phase very often. In these two worst cases, DFC respectively delivers 1.4X and 1.6X the performance of AC at 1 Gbps. The reasons is that, even in those cases, DFC shows 1.6X less L1-D cache misses compared to AC. Note, in the first case, they also frequently write to the memory to note the pattern IDs matched. However, these are very unlikely cases because often an upstream firewall can easily cut the flow based on the 5-tuple information for flows containing many malicious patterns or that try to attack the pattern matching system. Our results show that DFC outperforms AC both in the average and worst case scenarios.

Memory footprint: Figure 15 shows the memory footprint of AC and DFC including the pattern strings. We see that DFC occupies 3.1 to 4.8 times less memory compared to AC. AC requires 239.2 MB for 26K patterns from the ET-Pro ruleset. In contrast, DFC takes up only 59.4 MB memory for the same patterns. The memory footprint of the initial DF is negligible. The total footprint is actually dominated by compact tables that account for 95.0%. Progressive filtering takes up only 0.2%, and the string patterns take up 5.0% with 26k ET-Pro patterns.

Throughput with smaller CPU cache: An application performing multi-pattern matching may share CPU cache with other middlebox applications in some environments, such as network function virtualization (NFV). To evaluate whether the benefits of DFC still remain in such circumstances, we launch an additional thread that consumes half (10MB) of the L3 cache. Even in this case, DFC still outperforms AC by a factor of 1.9 with packet contents from real traffic and ET-Pro ruleset.

Initialization Time: Due to policy change and newly discovered attack, patterns used in middlebox applications are continuously kept updated. For example, new patterns for IDS are typically released each day [15]. To evaluate the overhead for updating data structures, we measured how long it takes to construct data structures for AC and DFC with ET-Pro ruleset.

As depicted in Figure 19, the gap between DFC and AC becomes larger; As the number of pattern increases from 1K to 50K, the speedup improves from 30X to 78X. It is because the initialization time of AC increases superlinearly while that of DFC increases linearly.

5.2 Applications of DFC

Intrusion detection: We apply DFC to an intrusion detection system. In particular, we use Kargus-CPU [39] which is a software IDS that is functionally compatible with Snort and provides multi-threading support and high-performance packet I/O. We measure the end-to-end performance of the IDS using synthetic/real traffic workloads and ET-Pro ruleset except rules that do not contain string fields. Note, this is strongly recommended practice [23]. For synthetic traffic, we use randomly generated the packet payload. For real traffic workload, we replay the packets captured from a commercial cellular ISP as fast as possible to attain a peak transmission rate of up to 70 Gbps. Up to 2 machines are used to generate the traffic. Because the workload consists of real-life flows, the flow management module takes up additional CPU cycles to update per-flow state. The patterns are also grouped based on the 5-tuple flow information. The IDS performs flow reassembly and feeds in the stream for pattern matching. This is representative of how IDSs actual work in a real environment.

Figure 16 shows the performance with Snort VRT ruleset for synthetic traffic by varying packet sizes. For large packets ($\geq$ 512B) DFC improves the performance by a factor of up to 2.6. For 128B packets, the difference is small because packet processing overhead is far greater than that of pattern matching since the amount of payload is only 74B per packet. Figure 17 shows the result while varying the number of patterns from 1K to 20K from the ET-Pro ruleset. DFC shows 120% improvement in performance over AC with 10K patterns. Even though some fraction of the CPU cycles are being spent in flow reassembly, DFC improves performance by 50 to 130%. Note that Kargus-GPU delivers a factor of 1.6 to 2.3 improvement over Kargus-CPU using two GPU cards.

Web application firewall: We apply DFC to an L7 Web firewall, ModSecurity. We benchmark the performance using the OWASP ModSecurity Core Ruleset [14]. Using

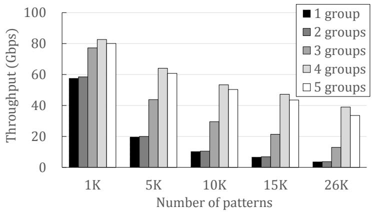


Figure 20: Performance for various number of classification

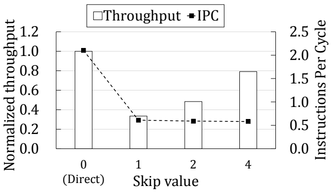


Figure 21: Filtering throughput and instruction throughput (IPC) of DF and "skip" heuristics

	DF	Bloom
Filtering throughput	1	0.42
Instructions/byte	1	2.5

Table 5: Normalized throughput and # of instructions per byte of text of DF and Bloom Filter using two light-weight rolling hashes [53]

ab [2], we request random files of size 10 KB. We choose 10 KB because it is the most frequently found Web object size according to HTTP Archive [11]. We vary the fraction of malicious requests from 0 to 100%. Note, ModSecurity inspects both the request and the response. When a malicious pattern is detected in the request, it generates a 403 forbidden response.

Figure 18 shows the transaction throughput per second. We report the performance result. DFC improves the performance by 90% with innocent request/responses. When the fraction of malicious requests increases, the transaction throughput goes up because response body is not inspected. For 100% malicious requests, pattern matching is only applied to requests, and the performance is dominated by the TCP processing overhead.

We also measure the performance with HTTP request/responses extracted from our real traffic trace where most of the traffic is HTTP. We modify ab and Apache to generate the same request and responses. The result shows that DFC delivers 57% improvement over AC.

Traffic classification: We apply DFC to HTTP traffic classification using nDPI [7], an open source DPI library. nDPI performs flow reassembly to gather fragmented HTTP headers, feeds them in pattern matching module that uses Aho-Corasick, and classifies which application generated the traffic by pattern-matching domain names in the header. We replace AC with DFC. We measure the performance of the traffic classification using the packets captured from a commercial cellular ISP and the top 100K domain names on Alexa [4]. Our result shows that DFC improves the performance by 60% (from 4.2 Gbps to 6.7 Gbps).

Anti-virus: We apply DFC to an anti-virus application, ClamAV [43]. ClamAV uses string and regular expression-like signatures that contain wildcard characters. The AC algorithm processes signatures containing wildcard characters and Wu-Manber (WM) handles fixed string signatures. We replace WM to use DFC and compare against SplitScreen that uses feed-forward Bloom filters [31] to improve ClamAV.

We use 1.5 million ClamAV signatures and scan a clean install of Microsoft Office 2015. For SplitScreen, we run both server and client on the same machine, while using the parameters from the paper [31]. Our result shows that DFC brings 75% the performance improvement (from 14.3 MB/s to 25.2 MB/s). When the files are malicious, the gap increases because SplitScreen must perform exact matching with AC using the output of FFBF, which is expensive. We also perform a standalone microbenchmark of FFBF and DFC using 1.5 million string patterns. The result shows that DFC's exact matching outperforms FFBF's inexact matching by a factor of 1.9. We see that DFC accesses memory (measured in terms of L1-data cache loads) 1.3 times less frequently, incurs 1.6 times less L1 cache misses, and uses 2.1X less instructions compared to FFBF.

# of patterns	**1K**	**5K**	**10K**	**15K**	**26K**
1B DF	41 Gbps	33 Gbps	28 Gbps	25 Gbps	21 Gbps
2B DF	**76 Gbps**	**55 Gbps**	**45 Gbps**	**37 Gbps**	**28 Gbps**
3B DF	66 Gbps	47 Gbps	37 Gbps	32 Gbps	25 Gbps

Table 6: Performance sensitivity to DF size (ET-Pro ruleset)

# of patterns	**1K**	**50K**	**1000K**	**1500K**	**2000K**
1B DF	14 Gbps	6 Gbps	5 Gbps	5 Gbps	5 Gbps
2B DF	**84 Gbps**	21 Gbps	18 Gbps	18 Gbps	18 Gbps
3B DF	53 Gbps	**49 Gbps**	**28 Gbps**	**25 Gbps**	**23 Gbps**

Table 7: Performance sensitivity to DF size (ClamAV ruleset)

5.3 Performance Contributions and Parameters

Performance contribution: Initial filtering and progressive filtering are very effective in screening innocent traffic. We measure the fraction of windows that actually reach the verification phase. We use real traffic workload of Table 4 and the ET-Pro ruleset without network I/O. With 26K patterns, 6.2% of windows reach progressive filtering. Progressive filtering further filters out up to 83.9% of windows. As a result, 4% reach the verification phase. A relatively large fraction of windows (8–14%) that arrives at the verification phase generates an exact match.

We evaluate the importance of the 2-stage hierarchy and progressive filtering by measuring the performance of DFC without each of them using real traffic and 100% malicious traffic. For the latter, we insert a malicious pattern for every packet in the real traffic workload.

Without progressive filtering, hash tables are looked up for verification whenever a window passes through the initial DF. In this case, the performance drops by 26% for the real workload and 24% for the malicious workload.

Our 2nd-stage hierarchy is especially beneficial when there are malicious patterns in the input, improving the worst case performance. Without the two-stage hierarchy, the performance drops down by 2.7X in the real traffic workload and 3.9X in the 100% malicious case. This is because finer-grained 2nd-state progressive filtering reduces the false positives in the average case, and reduces the collision and the number of patterns to be inspected in the final hash table in the malicious workload.

We now discuss parameters of DFC that affect its performance in the order of their importance.

Size of initial DF: We evaluate the effect of the size of the initial DF. We compare the performance of DFC by varying the size of the initial DF only. Table 7 and Table 6 show the result by varying the number of patterns for ClamAV and ET-Pro rulesets respectively. For ET-Pro, DF indexed by 2 bytes (2B DF) shows the highest performance across all cases under our workload. For ClamAV, DF indexed by 3 bytes (3B DF) peforms better when the number of patterns exceeds 50K. The difference is that ClamAV patterns are longer, which makes verification relatively more expensive. At the same time, as the number of patterns increases the verification phase becomes more expensive. Because 2B DF triggers verification more often than 3B DF, using 3B DF becomes more beneficial as the number of patterns increases.

Classification granularity: We quantify the effect of classification granularity. With finer-grained classification, the number of patterns inspected in the verification phase decreases at the cost of additional DF lookups in progressive filtering. We vary the number of classes, n, from 1 to 5 and measure the performance of DFC. Given n, we create pattern classes, $P_0, P_1, ..., P_{n-1}$. Figure 20 shows the performance comparison. As the number of patterns increases, the benefit of having a finer-grained classification becomes noticeable. This is because we can filter out the long patterns more effectively, if we can identify the pattern size with finer-grained classification. However, increasing the number of classes provides marginal benefit because the cost also increases. With 5 classes or more, the cost out-weights the benefit.

5.4 Comparison of Pattern Matching Primitives

The benefit of DFC comes from effectively leveraging the simple DF primitive. We benchmark the performance of our direct filter primitive in comparison with three other primitives used in existing algorithms: DFA's state transition, skip table, and hashing. [8]

DFA-based algorithms use state transition tables that cause frequent memory lookup and cache misses. Table 2 shows the number of memory references and cache misses for DF, DFC, and AC. DFC reduces memory accesses frequency by 1.8 times and L1 cache misses by 3.8 times compared to AC.

We compare the performance of DF and the "skip" table used for heuristics-based algorithms [71]. Note the skip approach has sequential data dependency—the next window to examine is dependent upon the value of the skip table entry—whereas DF lookups can be easily be pipelined. To quantify its performance benefit, we use a skip table that is indexed by the same two byte window as DF. Each table entry contains two bits that indicate how many bytes it can skip. We also increase the size of DF to 16 KB by using two bits for each entry. Thus, skip table and DF exhibit the same caching behavior. We set all skip values to be the same, but vary them from 1 to 4 across different experiments. When the skip value is four, the skip table is looked up on every *five* byte of input, whereas DF is looked up on every byte. Figure 21 shows the lookup performance of the two tables for our input. Surprisingly, DF is much faster than the "skip" approach even when the skip table performs five times less lookup (skip=4). DF's instruction throughput, measured in instructions per cycle, is also 3.5X times faster. This coupled with existence of many short patterns make the heuristics ineffective.

Finally, we compare the performance of a direct filter and a Bloom filter lookup that uses two light-weight rolling hash functions used in feed-forward Bloom filter (FFBF) [53].[9] We set the size of both data structures to the same value (8 KB). Table 5 shows that DF is 2.4 times faster than Bloom filter that performs hashing on every window. This is because hashing requires 2.5X more instructions and incurs more frequent memory accesses and cache misses due to additional memory lookups during the calculation of a rolling hash.

6 Related Work

Multi-pattern string matching: There have been a number of multi-string pattern matching algorithms, such as Aho-Corasick [24], Commentz-Walter [33] and Rabin-Karp [42] algorithms. Among these, AC is the most popular in security applications. For example, most software-based IDSs employ AC as the first-pass filter. For the traffic caught in the first phase, they perform perl compatible regular expression (PCRE) matching to confirm an intrusion against more sophisticated attack patterns. Since AC has to deal with all input traffic, its performance often determines the overall performance of an IDS in a normal situation. There have been optimization works that reduces the state transition table size by compressing DFA states [65], that splits AC tables into RAM and CAM [34], or that employs binary transition tables [64]. Some algorithms (e.g., SigMatch [41] and FFBF [53]) perform filtering at the cost of false positives. To support exact

[8] All results are measured on a machine with two Intel Xeon E5-2690 CPUs, using a randomly generated byte stream with 26K string patterns from the ET-Pro ruleset.

[9] The rolling hash we use [53] is known to be 2.5 times faster than a rolling hash used in the Rabin-Karp algorithm [32,42].

matching, these algorithms typically rely on traditional exact pattern matching.

Many proposals also leverage hardware support, including GPUs [39, 44, 67–69], FPGAs [62, 63], many-core processors [54], new ASIC designs [64], and network processors [49]. Kargus [39] implements a GPU-based Aho-Corasick engine that delivers 39 Gbps for randomly-generated payloads and 2.4K Snort rules with a NVIDIA GTX 580 GPU and an Intel X5680 CPU. Sourdis and Pnevmatikatos [63] present a FPGA-based string pattern matching that reduces the area cost by sharing comparators for different patterns. It achieves 9.7 Gbps of throughput with 1.5K Snort rules on Virtex2-6000. These studies focus on enhancing the performance of existing algorithms on hardware, whereas DFC presents a new algorithm that leverages the performance characteristics of modern CPUs. We believe that DFC can also benefit from hardware implementations and leave it as future work.

Regular expression matching: Many studies [26–28, 45, 46, 49, 51, 52, 56, 59] have proposed new NFA/DFA-based algorithms for regular expression matching. These studies predominantly use FPGA or ASIC to take advantage of the high degree of parallelism and on-chip memory for high performance [45]. The overarching goal is to create a single automaton that matches multiple regular expressions (regex). The core challenge is that a single automaton is fast, but space-inefficient because combining multiple regex into a single DFA causes state explosion [61]. In contrast, NFA-based solutions are slow, but require small amount of memory. Thus, many efforts [26, 45, 50] focus on reducing the memory footprint of DFA to utilize the fast on-chip memory as much as possible. XFA [60, 61] presents a design that tries to achieve the best of both worlds. It uses much less memory than using a single DFA and is faster than using multiple DFAs. But, the performance is still much slower (at least 6 times) than using a single DFA. Note, DFA-based regular expression matching is much slower than string matching. Due to the distinct trade off in performance and expressiveness, both string and regular expression matching are commonly used for deep packet inspection.

Advances in software-based packet processing: Advances in packet processing technologies [37, 38, 40, 57] are accelerating cloud-based middleboxes [58] and network function virtualization, which call for efficient primitives for fast deep packet inspection. Recently, G-opt [40] showed that CPU can deliver most of the benefits of GPU by proposing an element switching technique and hiding the memory latency. DFC accelerates pattern matching by avoiding dependencies between instructions and reducing memory access frequency, whereas G-opt proposes a generic technique for memory latency hiding. The two approaches are thus orthogonal. Our preliminary evaluation shows that DFC outperforms G-opt's hand-optimized pattern matching by 42 to 71% for Snort VRT ruleset, depending on the number of pattern groups to match. However, we believe applying memory latency hiding can further enhance DFC's performance (e.g., G-opt's efficient hash table lookup can be applied to DFC). We leave this as future work.

7 Conclusion

Middlebox services that inspect packet payloads have become commonplace. With the popularization of cloud-based DPI services, their performance and cost-efficiency have become important. However, for many DPI applications, pattern matching poses a severe performance bottleneck. This paper addresses an important problem of scaling the performance of multi-pattern string matching. Using cache-efficient data structures and progressive filtering based on the pattern lengths, we develop a high-performance string matching algorithm that works well across a variety of pattern sets and workloads. DFC classifies patterns based on the their size and applies filters progressively in multiple stages. Our evaluation shows that DFC improves state-of-the-art by 100 to 150% with a popular IDS pattern set and real traffic traces. We demonstrate that replacing existing pattern matching module with DFC results in a 57-160% performance improvement in four different applications.

Acknowledgement

We would like to thank our shepherd Paolo Costa and anonymous reviewers for their feedback and suggestions and Jaehyun Nam, Hyeontaek Lim and Iulian Moraru for their comments on initial design. We also thank Huy Hoang for his help on testing. This work was supported by IITP granted by the Korea government (MSIP) [B0126-15-1078, Creation of PEP based on automatic protocol behavior analysis and Resource management for hyper connected for IoT Services], [B0101-15-1368, Development of an NFV-inspired networked switch and an operating system for multi-middlebox services], and by NRF-2013R1A1A1076024.

References

[1] A developer's guide to complying with PCI DSS 3.0 Requirement 6. http://www.ibm.com/developerworks/library/se-pcireq6/index.html. [accessed 01-Sep-2015].

[2] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.2/en/programs/ab.html. [accessed 01-Sep-2015].

[3] Akamai Cloud Security Solutions: Kona Web Application Firewall. https://www.akamai.com/us/en/multimedia/documents/product-brief/kona-web-application-firewall-product-brief.pdf. [accessed 01-Sep-2015].

[4] Alexa - Actionable Analytics for the Web. http://www.alexa.com.

[5] Can the WAF help with a DDoS attack? https://support.cloudflare.com/hc/en-us/articles/200172116-Can-

the-WAF-help-with-a-DDoS-attack-. [accessed 29-Aug-2015].

[6] Capacity Planning for Snort IDS: Bilbous, Not Tapered. http://mikelococo.com/2011/08/snort-capacity-planning/. [accessed 01-Sep-2015].

[7] Configuring nDPI for custom protocol detection. http://www.ntop.org/ndpi/configuring-ndpi-for-custom-protocol-detection/. [accessed 01-Sep-2015].

[8] Emerging Threats Ruleset. http://rules.emergingthreats.net/.

[9] How The Great Firewall Works. http://cs.stanford.edu/people/eroberts/cs201/projects/international-freedom-of-info/china_2.html. [accessed 09-Sep-2015].

[10] How the NSA's domestic spying program works. https://www.eff.org/nsa-spying/how-it-works. [accessed 09-Sep-2015].

[11] HTTP Archive. http://httparchive.org.

[12] Intel Performance Counter Monitor - A better way to measure CPU utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[13] ModSecurity. https://www.modsecurity.org/.

[14] OWASP ModSecurity Core Rule Set. http://spiderlabs.github.io/owasp-modsecurity-crs.

[15] Proofpoint ET Pro - Ruleset The Expert Solution. http://www.emergingthreats.net/products/etpro-ruleset.

[16] Secure, fast, and easy Web Application Firewall. https://www.cloudflare.com/waf. [accessed 09-Sep-2015].

[17] Shorewall: iptables made simple. http://shorewall.net/.

[18] Snort Intrusion Detection System. https://snort.org.

[19] Snort Ruleset 2.9.7.0 from Snort Downloads. https://www.snort.org/downloads/. [accessed 01-May-2015].

[20] Snort User Manual 2.9.7. http://manual.snort.org/.

[21] Suricata: Open Source IDS. http://suricata-ids.org/.

[22] The Great Firewall of China: Keywords Used to Filter Web Content. http://www.washingtonpost.com/wp-dyn/content/article/2006/02/18/AR2006021800554.html. [accessed 09-Sep-2015].

[23] Writing Good Rules: Content Matching. http://manual.snort.org/node36.html#SECTION00491000000000000000. [accessed 01-Sep-2015].

[24] Aho, Alfred V. and Corasick, Margaret J. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM (CACM)*, 18(6):333–340, June 1975.

[25] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. Generating Realistic Workloads for Network Intrusion Detection Systems. *ACM Special Interest Group on Software Engineering (SIGSOFT) Software Engineering Notes (SEN)*, 29(1), January 2004.

[26] Zachary K. Baker and Viktor K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2004.

[27] Michela Becchi and Patrick Crowley. Extending Finite Automata to Efficiently Match Perl-compatible Regular Expressions. In *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2008.

[28] Michela Becchi and Patrick Crowley. A-DFA: A Time- and Space-Efficient DFA Compression Algorithm for Fast Regular Expression Evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):4:1–4:26, April 2013.

[29] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM (CACM)*, 13(7):422–426, July 1970.

[30] R.S. Boyer and J.S. Moore. A Fast String Searching Algorithm. *Communications of the ACM (CACM)*, 20(10):762–772, October 1977.

[31] S. K. Cha, I. Moraru, J. Jang, J. Truelovea, D. G. Andersen, and D. Brumley. SplitScreen: Enabling Efficient, Distributed Malware Detection. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[32] J. D. Cohen. Recursive Hashing Functions for N-grams. *ACM Transactions on Information Systems (TOIS)*, 15:291–320, July 1997.

[33] Beate Commentz-Walter. A String Matching Algorithm Fast on the Average. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, London, UK, UK, 1979.

[34] Vassilis Dimopoulos, Ioannis Papaefstathiou, and Dionisios Pnevmatikatos. A Memory-efficient Reconfigurable Aho-Corasick FSM Implementation for Intrusion Detection Systems. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2007.

[35] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Operational Experiences with High-volume Network Intrusion Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2004.

[36] Branislav Durian, Jan Holub, Hannu Peltola, and Jorma Tarhio. Tuning BNDM with Q-grams. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2009.

[37] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-Accelerated Software Router. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2010.

[38] Intel. Data Plane Development Kit (DPDK). http://dpdk.org.

[39] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: A Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[40] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the Bar for Using GPUs in Software Packet Processing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[41] Ramakrishnan Kandhan, Nikhil Teletia, and Jignesh M Patel. SigMatch: Fast and Scalable Multi-pattern Matching. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 3(1-2):1173–1184, 2010.

[42] Richard M Karp and Michael O Rabin. Efficient Randomized Pattern-matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[43] T. Kojm. ClamAV. http://www.clamav.net/.

[44] Lazaros Koromilas, Giorgos Vasiliadis, Ioannis Manousakis, and Sotiris Ioannidis. Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures. In *Proceedings of the ACM/IEEE symposium on Architectures for Networking and Communications Systems (ANCS)*, 2014.

[45] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced Algorithms for Fast and Scalable Deep Packet Inspection. In *Proceedings of the ACM/IEEE symposium on Architectures for Networking and Communications Systems (ANCS)*, 2006.

[46] Janghaeng Lee, Sung Ho Hwang, Neungsoo Park, Seong-Won Lee, Sunglk Jun, and Young Soo Kim. A High Performance NIDS using FPGA-based Regular Expression Matching. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2007.

[47] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf. [accessed 01-Sep-2015].

[48] Po-Ching Lin, Zhi-Xiang Li, Ying-Dar Lin, Yuan-Cheng Lai, and F.C. Lin. Profiling and Accelerating String Matching Algorithms in Three Network Content Security Applications. *IEEE Communications Surveys and Tutorials*, 8(2):24–37, 2006.

[49] Rong-Tai Liu, Nen-Fu Huang, Chih-Hao Chen, and Chia-Nan Kao. A Fast String-matching Algorithm for Network Processor-based Intrusion Detection System. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):614–633, 2004.

[50] Chad R Meiners, Jignesh Patel, Eric Norige, Alex X Liu, and Eric Torng. Fast Regular Expression Matching using Small TCAM. *IEEE/ACM Transactions on Networking (TON)*, 22(1):94–109, 2014.

[51] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast Regular Expression Matching using Small TCAMs for Network Intrusion Detection and Prevention Systems. In *Proceedings of the USENIX Conference on Security*, 2010.

[52] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for Accelerating Snort IDS. In *Proceedings of the ACM/IEEE symposium on Architectures for Networking and Communications Systems (ANCS)*, 2007.

[53] Iulian Moraru and David G Andersen. Exact Pattern Matching with Feed-Forward Bloom Filters. *Journal of Experimental Algorithmics (JEA)*, 17:3–4, 2012.

[54] Jaehyun Nam, Muhammad Jamshed, Byungkwon Choi, Dongsu Han, and KyoungSoo Park. Haetae: Scaling the Performance of Network Intrusion Detection with Many-Core Processors. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[55] Marc Norton. Optimizing Pattern Matching for Intrusion Detection. *Sourcefire, Inc., Columbia, MD*, 2004.

[56] Jignesh Patel, Alex X Liu, and Eric Torng. Bypassing Space Explosion in High-speed Regular Expression Matching. *IEEE/ACM Transactions on Networking (TON)*, 22(6):1701–1714, 2014.

[57] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.

[58] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2012.

[59] Reetinder Sidhu and Viktor K. Prasanna. Fast Regular Expression Matching Using FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2001.

[60] Randy Smith, Cristian Estan, and Somesh Jha. XFA: Faster Signature Matching with Extended Automata. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2008.

[61] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2008.

[62] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. *Field Programmable Logic and Application (FPL)*, pages 880–889, 2003.

[63] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded CAMs for Efficient and High-speed NIDS Pattern Matching. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[64] Lin Tan and Timothy Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005.

[65] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic Memory-efficient String Matching Algorithms for Intrusion Detection. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2004.

[66] Chris Ueland. Scaling cloudflare's massive waf. http://www.scalescale.com/scaling-cloudflares-massive-waf/. [accessed 09-Sep-2015].

[67] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2008.

[68] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P Markatos, and Sotiris Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2009.

[69] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: A Multi-parallel Intrusion Detection Architecture. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[70] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *Proceeding of the ACM International Conference on Mobile systems, applications, and services (MobiSys)*, 2013.

[71] Sun Wu and Udi Manber. Fast Text Searching: Allowing Errors. *Communications of the ACM (CACM)*, 35(10):83–91, October 1992.

Diplomat: Using Delegations to Protect Community Repositories

Trishank Karthik Kuppusamy Santiago Torres-Arias Vladimir Diaz Justin Cappos
Tandon School of Engineering, New York University

Abstract

Community repositories, such as Docker Hub, PyPI, and RubyGems, are bustling marketplaces that distribute software. Even though these repositories use common software signing techniques (e.g., GPG and TLS), attackers can still publish malicious packages after a server compromise. This is mainly because a community repository must have immediate access to signing keys in order to certify the large number of new projects that are registered each day.

This work demonstrates that community repositories can offer compromise-resilience and real-time project registration by employing mechanisms that disambiguate trust delegations. This is done through two delegation mechanisms that provide flexibility in the amount of trust assigned to different keys. Using this idea we implement Diplomat, a software update framework that supports security models with different security / usability tradeoffs. By leveraging Diplomat, a community repository can achieve near-perfect compromise-resilience while allowing real-time project registration. For example, when Diplomat is deployed and configured to maximize security on Python's community repository, less than 1% of users will be at risk even if an attacker controls the repository and is undetected for a month. Diplomat is being integrated by Ruby, CoreOS, Haskell, OCaml, and Python, and has already been deployed by Flynn, LEAP, and Docker.

1 Introduction

Community repositories, such as Docker Hub [32], Python Package Index (PyPI) [66], RubyGems [68], and SourceForge [78] provide an easy way for a developer to disseminate software. These repositories are run by a central group of administrators and distribute third-party software for hundreds of thousands of projects. Unlike traditional repositories, the administrators of community repositories do not dictate which projects can or cannot be hosted; instead, developers are free to curate their own projects. Community repositories are immensely popular and collectively serve more than a billion packages per year. Unfortunately, the popularity of these repositories also makes them an attractive target to attackers.

Attacks on community repositories are unfortunately a common occurrence that threaten users who rely on their software. Major repositories run by Adobe, Apache, Debian, Fedora, FreeBSD, Gentoo, GitHub, GNU Savannah, Linux, Microsoft, npm, Opera, PHP, RedHat, RubyGems, SourceForge, and WordPress repositories have all been compromised at least once [4,5,7,27,28,30,31,35,36,39–41,48,59,61,62,67,70,79,80,82,86,87,90]. For example, a compromised SourceForge repository mirror located in Korea distributed a malicious version of phpMyAdmin, a popular database administration tool [79]. The modified version allowed attackers to gain system access and remotely execute PHP code on servers that installed the software. This is despite the use of off-the-shelf solutions like TLS and GPG, which (for reasons described in Section 4) are known to be ineffective against practical threats in this domain. For example, we found that, on PyPI, so few developers sign packages and so few users download signatures that within a one month period there was not a single user who downloaded only GPG-signed packages and their signatures.

Prior work has shown that *delegations* [1, 52, 92] help the users of a repository remain secure even if it is compromised [71]. Delegations add security to repositories when the root of trust is an *offline key*, such as a key stored on a disconnected server that must be manually used. Although using offline keys works for software repositories that have infrequent release cycles, community repositories commonly register dozens of new projects daily, with new packages uploaded every few minutes. As such, it is not practical to require manual operations for project registration.

This paper presents Diplomat, a practical security system that provides a community repository with immediate project registration and compromise-resilience. Our key insights come from delegation techniques that utilize multiple online and offline keys to take advantage of the best properties of both. Central to this strategy is the use of a *prioritized delegations* [44, 56–58] mechanism for disambiguating trust statements. Prioritized delegations enforce an order among parties who would otherwise be equally trusted. In addition, our work uses *terminating delegations*, which prevent statements by less trusted parties from being trusted for a package. The combination of prioritized and terminating delegations allows an offline key's attestation about a project to be trusted over information provided by an online key. Placing greater trust in the offline key provides compromise-resilience because an attacker who compromises the repository cannot modify a package without being detected. However, the online key may still be used (and trusted) to create new projects.

We feel one of the main contributions of this work is how it balances security and usability to solve a practical, widespread problem. The security models and experiences we describe in this work are based upon practical lessons learned from ongoing integrations with RubyGems [75–77], Haskell [91], CoreOS [64], and OCaml [38] and production use in Flynn [65], LEAP (Bitmask) [53], and Docker [63].

Contributions.

- We examine threats to community repositories and find that current security approaches inadequately address these threats. In particular, these techniques are unable to accommodate both compromise-resilience and instant registration of new projects.

- We use two types of delegations — prioritized and terminating delegations — to design and implement Diplomat, the first security system that achieves both compromise-resilience and instant registration of new projects.

- We discuss two different security models — legacy and maximum — that provide slightly different usability / security trade-offs. Drawing on practical experience, we discuss procedures for managing offline key storage, usability for users and developers, recovering from compromises, and procedures to minimize the effort required of repository administrators.

- We evaluate the effectiveness of Diplomat using requests to PyPI, the main Python community repository. Our findings demonstrate that Diplomat will protect over 99% of PyPI's users, even if an attacker controls PyPI and is undetected for a month.

2 Background

We first discuss and define community repositories, paying attention to how they differ from traditional software repositories. We then provide some background on roles and delegations, two techniques used in compromise-resilient repositories [20–23, 71] that we will leverage to build Diplomat.

2.1 Community Repositories

A *community repository* hosts and distributes third-party software. Three groups of people, administrators, developers and users, interact with a community repository. The administrators, who are usually volunteers, manage the community repository software and hardware. Developers upload software to the repository, which is requested by users. Users install and validate software using a package manager, which downloads software through middlemen, such as content delivery networks and / or mirrors to reduce bandwidth costs.

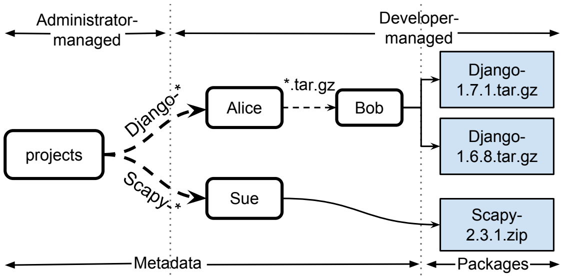


Figure 1: An example of delegation of trust in a software repository. The top-level `projects` role delegates to Sue for the Scapy project and Alice for Django. Alice further delegates to Bob the ability to create tar.gz packages for Django.

The software that is uploaded by developers is organized as follows. A developer registers a *project* with a unique name and adds access to other developers that work on the project. When a specific version of the software for that project is ready to be released, the software is built into a *package* (e.g., `Django-1.7.tar.gz`) and one of the developers uploads that package to the community repository. The community repository also distributes metadata about projects and packages (such as a list of package names) and includes metadata created by developers (such as a signature for a package).

2.2 Roles

One of the key security concepts used in compromise-resilient software repositories is that of a *role* [71]. A role defines the set of actions that a party is allowed to perform. For example, the `projects` role is trusted to sign metadata that indicates which developer keys belong to a project. Similarly, the `release` role is trusted to sign metadata that indicates which versions of each package and metadata are in the latest release. However, if the `release` role's key is used to sign the metadata that indicates which developer key belongs to a project, that signature will not be trusted because the key is not trusted for that role. This paper describes techniques that apply to the `projects` role's use of delegations, so the paper will focus primarily on this role.

2.3 Delegations

The use of *delegations* is a powerful strategy that has successfully been used in a variety of contexts, including distributed systems [92], role-based access control [73], trust management [16], delegation logic [57], and software repositories [20–23, 71]. In the context of software repositories, delegations are specifically used to distribute permissions to sign packages across different administrators and developers. If A can sign a package K, then A can delegate this permission to B so that B can sign K on behalf of A. The delegation is an indirect package signature, where B "speaks for" [52] A about K.

Although the `projects` role may sign packages because it is the root of trust for all packages, in Figure 1 the `projects` role has instead *delegated* the Django project (or the package path `Django-*`) to the public keys belonging to the developer Alice. Similarly, the Scapy project has been delegated to Sue. A delegation is simply a trusted map of which developer keys are responsible for signing which projects (or sets of packages). Based on this delegation, users would trust only Alice's signature on a Django package. Developers can further delegate entrusted packages to other developers. In this case, Alice has delegated some packages (any package matching the path `Django-*.tar.gz`) to the developer Bob. Thus, Bob speaks for Alice for only the `Django-*.tar.gz` packages, whereas Alice's signature on `Django-1.7.1.exe` (not shown) would be trusted instead of Bob's.

3 Threats and Threat Model

There are many risks that users of software repositories face. Attackers can interject traffic by proxy interception attacks [43], target weaknesses in TLS [37, 85], set up a malicious mirror [21, 79], exploit weaknesses in the network infrastructure [19, 81], compromise signing keys due to weaknesses [45, 84], or steal keys outright by exploiting a security vulnerability [25]. Furthermore, attackers have proven adept at compromising the repository or signing infrastructure of many companies. This leads us to consider a threat model where a compromise of at least some part of the system occurs.

3.1 Threat Model

We assume that an attacker can:

1. Compromise a running repository and / or any keys stored on the repository, including those situations where the key itself is unknown (e.g., due to hardware protection) but where the attacker is nevertheless able to sign malicious packages using the key [67].

2. Respond to user requests. This can be done either by acting as a man-in-the-middle, or compromising the repository or one of its mirrors.

An attack will be successful if the attacker can change the contents of a package that a user installs (e.g., to insert a backdoor [27, 41, 43, 61, 62, 67]). Existing software update systems protect against a wide array of other attacks such as replay and mix-and-match attacks [20–23, 71]. We protect against those attacks by leveraging the role and delegation layout from these prior works. Thus, those types of attacks are only briefly discussed in this paper, so that we may focus on key compromise resilience while allowing online registration of projects.

We assume that projects have trustworthy developers and that these developers, who store their keys external to the community repository, take measures to secure them. If a key corresponding to a project is compromised, we consider a community repository's security to be effective if it limits the impact of an attack to the project whose key has been compromised.

4 Analysis of Current Systems

In this section, we examine four security approaches that are used on repositories. These techniques allow administrators of community repositories to sign packages — either themselves or by delegating packages to their respective project developers. These security models are illustrated in Figure 2. These models are discussed in turn in the following subsections.

4.1 Existing Security Models

(a) Repositories sign with online keys

In community repositories such as PyPI, RubyGems and npm, all packages are signed only by repositories with online keys (Figure 2(a)). A repository may sign packages with a transport mechanism such as TLS or CUP [42]. These private keys are kept online because community repositories must publish new projects and packages as soon as possible. Unfortunately, because the key is online, a compromise of the repository would instantly render all packages vulnerable. For example, the npm community repository reported that a programming bug not only leaked its TLS keys, but also allowed attackers to remotely rewrite packages [90]. Since developers do not sign their packages in this security model, users who subsequently request packages that have been tampered with trust the repository's package signatures without question. This is because the transport mechanism is useful only for establishing the identity of the repository, but not the authenticity of the packages themselves as belonging to their respective developers.

(b) Developers sign with offline keys

Some community repositories, including PyPI and RubyGems, permit developers to sign their packages with offline GPG [83] or RSA keys before uploading them to the repository (Figure 2(b)). Unlike the previous security model, signatures are used to verify the authenticity of packages, not to authenticate the repository's identity. In this model, users must discover the correct key for a developer from an out-of-band channel and then use this to verify packages.

One substantial problem with this model is that finding

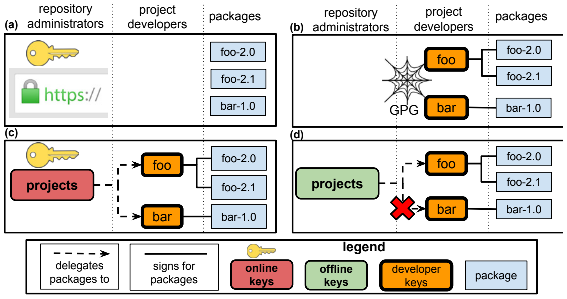


Figure 2: Existing security models for community repositories.

Roles and Responsibilities	
root	The root role is the locus of trust. It indicates which keys are authorized for the projects, release, and timestamp roles. It also lists the keys for the root role itself.
projects	The projects role is trusted to validate packages. Often, the projects role will delegate trust for a project to the responsible developers.
release	The release role indicates the latest versions of all metadata on the repository. This prevents a user from later being deceived into installing an outdated package.
timestamp	The timestamp role is responsible for indicating if the repository contents have changed. This role will often be performed by external parties, such as mirrors.

Figure 3: The top-level roles used within Diplomat.

and verifying developer keys remains a manual process, with the burden placed on both developers and users. Finding true developer keys can be tricky, especially with attackers distributing fake keys, as was the case with the Tor project [24]. The repository is compromise-resilient only inasmuch as users have found and verified the correct developer keys. While PyPI and RubyGems support this model, only 4% of PyPI projects even list a signature. Moreover, in a month long trace of package requests to PyPI, only 0.07% of users downloaded these signatures for verification. If signatures are not used, then attackers who compromise the repository may modify any package in whatever manner they choose.

Thus, repository administrators across diverse community repositories are seeking a better solution [33, 47, 91]. To quote the RubyGems security guide [69]: "The goal is to improve (or replace) the signing system so that it is easy for authors and transparent for users."

(c) Repositories delegate to projects with online keys

In this security model (Figure 2(c)), the `projects` role for a delegation framework like TUF [71] is signed with an online key. In order to solve the problem of which developer keys map to which packages, repositories will *delegate* a project (its set of packages) to the public keys of the developers of that project. For example, PyPI will delegate *all* packages of the Django project (matching, say, the package path `Django-*`) to the public key of the lead developer of the project who, in turn, may delegate the Django packages to other developers. Since the `projects` role key is online, a new project can be immediately registered by the repository, through a new delegation to a project.

This model does not build compromise-resilient community repositories precisely because the `projects` role can be compromised by an attacker. (This is true despite the fact that developers sign their respective packages with offline keys.) This is because the keys for the `projects` role are kept online. Thus, once an attacker has compromised a repository, he (or she) is free to rewrite delegations using the online private keys. Then, an attacker can have the `projects` role delegate trust for the `Django-*` packages to a key that the attacker controls. As such, the attacker could deceive users into installing malicious packages that did not originate from the project's developers.

(d) Administrators delegate to projects with offline keys

Unlike the previous TUF security model, which delegates trust using an online key, administrators could alternatively choose to delegate using offline keys (Figure 2(d)). This means that the `projects` role key (kept offline) delegates projects to developer keys. Therefore, this model does indeed build compromise-resilient repositories because attackers cannot rewrite delegations (and thus packages) after a repository compromise. The attacker's capabilities are limited to preventing clients from seeing new packages in a timely manner (freeze attack) or providing new package updates out of order (mix-and-match attack) [71]. This model is used by traditional repositories, including LEAP [53]. Unfortunately, this model is impractical to use in community repositories because new projects, which are created dozens of times a day, cannot be registered without an administrator using an offline key.

5 Diplomat: Architecture and Delegations

This section describes the architecture of Diplomat, a security system designed to allow community repositories to have both compromise-resilience and immediate project registration. It begins with a high-level overview that explains the roles and use of delegations within Diplomat (Section 5.1). Following this, we present two problems that a delegation-based system would face when used on community repositories (Section 5.2). Diplomat addresses these problems using two types of delegations: prioritized delegations (Section 5.3) and terminating delegations (Section 5.4). In the next section, (Section 6) we will demonstrate how to use these delegations to provide compromise-resilience so that even if online keys for project registration are stolen, projects that were previously registered (with offline keys) are not at risk.

5.1 Roles and Delegations in Diplomat

Much like our earlier work on TUF [71], Diplomat separates trust between different parties using the four top-level roles shown in Figure 3: `root`, `timestamp`, `release`, and `projects`. Each role produces metadata that fulfills a specific purpose. The `root` role specifies the public keys of the top-level roles (including its own) and can revoke the other top-level role keys, if needed. The `release` role indicates the latest version numbers of all Diplomat metadata (other than `timestamp`) that is available on the repository. The `timestamp` role references the latest `release` role metadata and will signify the last time the contents of the repository have changed. The `projects` role lists the available projects and either provides cryptographic hashes of packages or delegates trust to keys that provide those hashes. Each top-level role is only trusted for its assigned responsibilities; this minimizes the impact of a compromised role.

Our focus in this work is on the `projects` role (and the delegations it makes to non-top-level roles). Hence, details about any top-level role other than the `projects` role are only discussed as needed. Documentation is available that provides a more holistic discussion about the roles and their use [49, 50, 71].

The `projects` role is the root of trust for all packages on the repository; if a user wishes to download (and install) some package, he or she must first download and verify the latest `projects` role metadata. The user has the keys for this role because these public keys are contained within the `root` metadata file. The top-level projects role may delegate to other developers or vendors, which may also then delegate to others. A client can validate a package by following the chain of delegations until they find a trusted developer's metadata that contains the cryptographic hash of the package.

5.2 Problems With Delegation Ambiguity

Security problems can occur when a party cannot effectively control how much trust they place in a party when performing a delegation. To illustrate the problem, Figure 4 provides an example we will use throughout this section. In this example, A is the root of trust for packages. A delegates trust in any package with a name that matches the package path `bar-*` to B, and all packages (including `bar` packages) to C. This example illustrates two problems.

The Ordering Problem. Suppose that A has delegated `bar` to B and all packages to C. If B and C provide different cryptographic hashes for `bar-1.0`, which hash should be trusted?

Note that there is no "correct" resolution to this question, because A's intent is not clear. In some cases, the more specific delegation of `bar-*` to B should be trusted over the more general delegation of `*` to C. However, in other cases the reverse should be true. (In fact, the maximum security model in Section 6.1 has a general delegation which is prioritized over a specific delegation, which in turn is prioritized over another general delegation.) A solution must allow a party to express the intended order in which to resolve delegations.

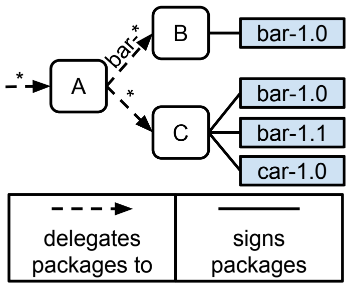


Figure 4: An example of ambiguous delegations. The label on a delegation specifies what packages to delegate.

The Failover Problem. Suppose that A wants B to be the only trusted party for `bar` packages, but C can be trusted with any other package. How can this be expressed?

There are clearly cases where failover is desirable (e.g., allowing a second developer to sign a package if the first does not) and those where it is not desirable (e.g., the ability for C to provide `bar-1.1`, if B is meant to be the only source of `bar` packages). A solution must enable the delegator to specify their intended behavior.

5.3 Prioritized Delegations

Diplomat uses *prioritized delegations* to order delegations between different parties and address the ordering problem. The key concept is to prioritize delegations based upon the order they occur in the metadata file. (This is similar to the manner in which firewall rules are processed in the order they are listed [60].) By exploiting the order in which delegations are listed, then the first delegation will be used before the second delegation, and so forth. For example, if B is listed before C in Figure 4, then the user would trust B over C for the `bar-1.0` package.

In case a role both delegates and signs a package, then the role's package signature takes precedence over its delegations. So if A signed the `bar-1.0` package, then A would be trusted for the package despite its delegation of the package to both B and C.

5.4 Terminating Delegations

Diplomat uses the concept of a *terminating delegation* to address the failover problem by halting the processing of delegations at a specified point. Terminating delegations instruct the client not to consider future trust statements that match the delegation's pattern. This stops the delegation processing once this delegation (and its descendants) have been processed. (Handling this case is conceptually similar to the use of the cut operator in Prolog [18] to stop computation, except that Diplomat uses this technique for security instead of efficiency.) A terminating delegation for a package causes any further statements about a package that are not made by the delegated party or its descendants to be ignored.

5.5 Processing Delegations

The algorithm for resolving delegations through the application of prioritized delegations involves a pre-order, depth-first search of the `projects` metadata. This algorithm is used on a client device when a user instructs the package manager to install a package.

To install a package, a recursive algorithm begins at the `projects` role and searches for the package of interest. First, all of the hashes in the metadata file provided by the `projects` role are checked to see if the requested package is listed. This is the "pre-order" check to see if the current party has information about the desired package. Following this, delegations are examined in their order of priority (i.e., the order they are listed). If a delegation selects a portion of the namespace to delegate (e.g., `bar-*`), then the algorithm ignores this delegation if the pattern does not match (e.g., if the request was for `foo-1.0`). For any matching delegation, the algorithm will (in order) recursively search for the package of interest. It does so by repeating the preceding steps on the highest priority delegatees (in order). If any of the delegations is a terminating delegation, then the algorithm is terminated at that point, even if this terminating delegation does not return with an answer, preventing further delegations from being considered.

6 Diplomat Security Models In Practice

This section describes how to set up Diplomat metadata to provide real-time project registration and compromise-resilience. To exemplify how Diplomat is used in practice, we describe two security models that have been standardized for use within the Python community: the maximum security model (Section 6.1) [50] and the legacy security model (Section 6.2) [49]. After describing these two models, we elaborate on other usability aspects of Diplomat, such as handling key compromises, setting up roles, and maintenance in Section 6.3.

The legacy and maximum security models provide different trade-offs for the security and the availability of packages for projects we call *rarely updated*. A rarely updated project is one for which its developers have not provided a signing key, often because the package is not actively maintained. Nevertheless, its packages may be actively downloaded by users. An example is the BeautifulSoup project on PyPI, which last released package version 3.2.1 on February 16th, 2012, but nonetheless was downloaded more than a hundred thousand times in January 2016. The maximum security model uses an offline key to sign these projects. If an update is made, users will not receive it until the repository administrators sign the package with an offline key. In contrast, the legacy security model (Section 6.2) handles rarely updated projects by signing them with the online `unclaimed-projects` role. Due to the fact that online keys are used, developers can immediately update unclaimed projects. However, this comes at the cost of leaving their users vulnerable in the event of a repository compromise. Thus, the maximum model provides higher security but delayed availability of new packages for rarely updated projects, whereas the legacy model provides exactly the opposite trade-off. The security analysis of these models is available in our technical report [51].

6.1 Maximum Security Model

The maximum security model [50] (Figure 5) aims to reduce the risk to users if the repository is compromised by an attacker at the cost of making users wait before retrieving a new package for a rarely updated project. The top-level `projects` role of the maximum security model delegates to three other roles. The first and highest priority delegation, `claimed-projects`, is assigned to projects who have developers sign their own project metadata with their own offline key. The next highest priority delegation, `rarely-updated-projects`, requires repository administrators to delegate these projects with a terminating delegation, and to sign for these packages with an offline key. Finally, the lowest priority delegation, `new-projects`, is targeted to new projects, which are signed by an online key. If an attacker compromises the repository, they can change the metadata that indicates which key should be trusted for new projects (and thus can forge packages for those projects), but due to the higher-priority, terminating delegations to existing projects, whether rarely updated or not, cannot modify those packages without being detected.

The highest priority delegation issued by the `projects` role is to the `claimed-projects` role. The `claimed-projects` role signs a terminating delegation of all packages of a project (such as `foo` or `flibble`) to the public keys of its developers. Projects may choose to delegate trust to developers' public keys, and either a project or a developer will sign and upload metadata about their packages. The use of a terminating delegation ensures that if a user attempts to verify a `foo` package, then the user would only search for the package among its developers. Most importantly, the `claimed-projects` role signs its delegations to projects with *offline* keys so that attackers cannot tamper with the packages of these projects after a reposi-

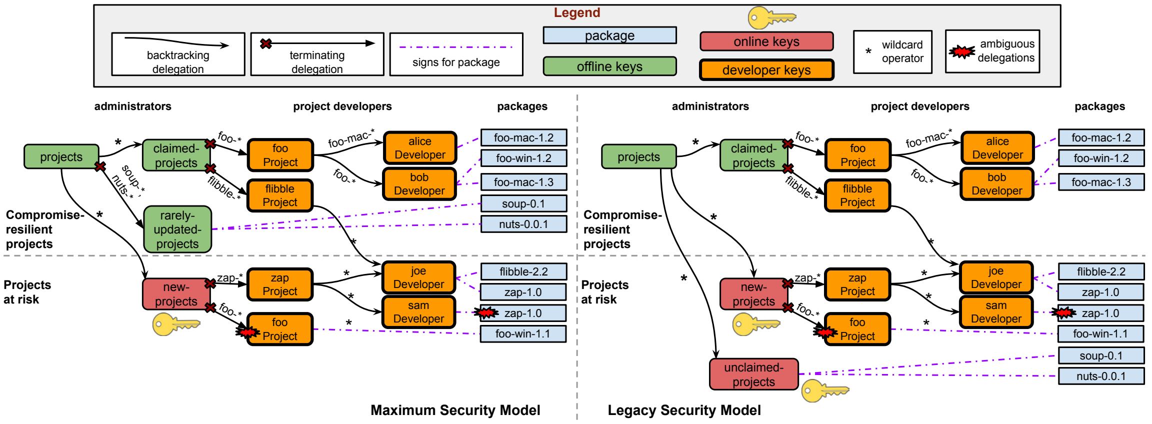


Figure 5: Maximum and legacy security models for community repositories. The red symbol indicates delegations that are not used due to earlier trust statements. Delegations that are higher on the figure (toward the top of the page) have higher priority.

tory compromise (without also compromising the private keys used by claimed project developers). However, since the delegation from the projects role to the claimed-projects role allows backtracking (i.e., it is not a terminating delegation), any requests for projects unknown to the claimed-projects role will not be terminated at this role, and will instead continue with the rarely-updated-projects role.

The second-highest priority delegation pertains to the rarely-updated-projects role. This role directly signs, with *offline* keys, all packages of rarely updated projects. Since the key used is offline, packages cannot be signed by this role without an action by the repository administrators. This delays the release of new packages of rarely updated projects.

The delegation from the projects role to the rarely-updated-projects role is a terminating one. Furthermore, the rarely-updated-projects delegation specifies only the package paths of rarely updated projects (such as soup-* and nuts-* in Figure 5). Because of this, no backtracking is performed to search elsewhere for the package signatures of a project already delegated to this role.

Finally, the new-projects role is able to assign keys to package names that were not already defined. This role is served by an online key that delegates trust to newly created projects. However, since the role has an online key, there is a substantial risk of compromise. By assigning this role the lowest priority (and using prioritized, terminating delegations for claimed and rarely updated projects), an attacker will be able to only impact newly-created projects if the repository is compromised. For example, in Figure 5, the new-projects role's second delegation of foo is ignored due to the first terminating delegation of foo having a higher priority delegation via the claimed-projects role.

6.2 Legacy Security Model

The legacy security model [49] (Figure 5) is very similar to the maximum security model, but differs in the way that it handles rarely updated packages. This model allows new packages for rarely updated projects to be available immediately, while still providing security benefits to the users of claimed projects.

Like the maximum security model, the legacy security model includes the claimed-projects and new-projects roles. However, in the legacy security model, the repository uses an online key to sign for *unclaimed* projects. Like rarely updated projects, unclaimed projects are also signed by the repository instead of developers, but with the unclaimed-projects role that uses online keys. The unclaimed-projects role has the lowest priority delegation and, since the key is online, all projects signed with this key are at risk in the event of a compromise. Prioritized and terminating delegations of claimed projects signed by the claimed-projects role ensure that, even when the repository is compromised, packages of claimed projects are not at risk. Thus, all packages of unclaimed projects—unlike rarely updated projects—are available immediately, but vulnerable in case of a repository compromise (just like new projects).

The legacy model is drawn from our integration and deployment experience with the Python and Docker community repositories. Both repositories wanted to allow the repository to sign packages on behalf of developers who did not wish to do so. However, since its key is stored offline, using the rarely-updated-projects role would prevent administrators from quickly releasing new packages from these developers. The unclaimed-projects role permits the repository to immediately sign packages on behalf of these developers.

Diplomat enables a repository to smoothly transition from the legacy to the maximum security model. The repository administrators can first have the projects role delegate to both the rarely-updated-projects and unclaimed-projects roles. The administrators can then move projects from the unclaimed-projects to the rarely-updated-projects role (Section 7.2.2), and / or require developers to register a project key to upload a new package. Either way, project developers will be incentivized to transition out of the unclaimed-projects role over time (using policies explored in Section 7), improving security.

Docker Hub uses a similar security model for its deployment of Diplomat. As of February 2016, Docker was signing the most popular projects, such as Ubuntu (a policy we explore in Section 7.2.1). Docker plans to explore options such as incentivizing project developers to sign their packages (by visually distinguishing or showing signed packages first in search results on Docker Hub), or requiring developers to sign packages in order to upload a new one (a policy we explore in Section 7.2.3).

6.3 Using Diplomat

Regardless of whether a repository uses the legacy or maximum security model, Diplomat requires essentially the same actions by users, developers and administrators.

Users. End users that install software through a package manager that uses Diplomat do not need to perform any actions and see no difference in their package manager's behavior. This is because Diplomat downloads and verifies its metadata before the package manager is allowed to install a package. The delegation structure in Diplomat manages keys on behalf of the user, which avoids the issues involved with locating and downloading appropriate project keys (e.g., the model in Figure 2(b)). The only situation where the user will be aware of Diplomat's existence is when a repository was compromised and it produces a message that notifies that signatures on data provided by the repository do not match.

Developers. To use Diplomat to protect a project, a developer must create a public / private key pair and upload the public key to their community repository. The repository will associate that key with the project first through the new-projects role and later through the claimed-projects role. If the project's leaders elect to do so, they may further delegate trust to different members of the project, who may also sign packages so that the project key need not be shared. Whenever a package is released, a developer must also generate a piece of signed Diplomat metadata, the format of which is in our standards document [49], that provides the cryptographic hash of the package. The actions needed to create or update this metadata can be added to the project's packaging scripts so that it is performed automatically when a new package is built. Diplomat provides a set of command-line tools [88] that helps developers to perform and automate those actions.

If the project key (i.e., the key that is delegated to directly by the claimed-projects or new-projects role) is compromised, the repository administrator will need to perform an action (discussed below) before trust in this key is revoked. However, if an individual developer key is compromised, the project can simply sign and upload a new piece of metadata that changes the key or removes that delegation. This action does not involve repository administrators.

Repository Administrators. Most of the work involved with using Diplomat comes from the initial setup. Repository administrators need to generate the keys for the roles and set up the initial delegations in their metadata. Offline keys should be stored in one or more devices that are not network connected and high value roles should require signatures from multiple keys. (We discuss procedures for this in more detail in the standards documents [49, 50].) The repository software needs to be modified so that Diplomat metadata is generated and updated whenever projects are registered or packages are uploaded. Diplomat provides administrators with command-line tools and APIs [89] that automate these actions and make it easy to integrate them into an existing repository.

Periodically (e.g., every few weeks), administrators will perform a maintenance operation on the repository to help it remain resilient to a key compromise. The administrators should append the new-projects role metadata to the claimed-projects role metadata and sign the resulting metadata with the claimed-projects key. If the rarely-updated-projects role exists, then newly uploaded packages that are not signed by their developers should be added. Revoked project keys, which are discussed below, are also replaced. Once this updated metadata file is uploaded, this makes it so that an attacker who compromises the repository cannot replace the key for any projects included before that point. Administrators will also calculate the cryptographic hash of every package on the repository and store this data on an offline system. This allows administrators to have a known-good hash of each package to detect and recover from a repository compromise.

Securely revoking a project key. When an authorized party wants to revoke trust in a project key, they notify the repository administrators and undergo an identity verification procedure [6]. (The exact procedure depends on the deployment and is out of the scope of this paper.) Once this is done, the administrators will write the new project key into a revoked role metadata file (not retrieved by users). When the maintenance operation is performed to generate the new claimed-projects file, the revoked keys are replaced. Administrators may publish revoked project keys to Twitter as both a notification service and as a way of having a public log of project keys that will change in the next maintenance operation.

Securely recovering from a repository compromise. When a repository compromise has been detected, the integrity of three types of information must be validated. First, the keys for the `new-projects` and `unclaimed-projects` roles of the repository need to be revoked because they may have been compromised (i.e., their online keys have been compromised). The metadata for these roles must be discarded or returned to a known-good state. These keys can be revoked by having the offline `projects` role key sign new role metadata that delegates to a new key.

Second, the role metadata of the repository may have been changed. Metadata signed by the top-level `timestamp` and `release` roles may have been changed, enabling the attacker to launch mix-and-match and freeze attacks [20–23]. These keys should be revoked by the top-level `root` role, as is discussed in our prior work [71].

Third, the packages themselves may have been tampered with. Packages that existed the last time the `claimed-projects` and `rarely-updated-projects` role files were signed, can be verified using the stored hash information. Also, new packages that are signed by developers with the `claimed-projects` role may be safely retained. However, any package signed by developers using the `new-projects` or `unclaimed-projects` role should be discarded.

7 Evaluation

In order to better understand to what extent Diplomat makes community repositories compromise-resilient, we investigated the following questions:

- Does using the security models in Diplomat improve users' security in the event of a repository compromise? Is Diplomat better than existing solutions like TLS or GPG signatures? (Section 7.1)
- Suppose that the legacy security model is adopted for usability reasons. If the goal is to maximize security, what strategy should be used to get projects to sign packages? For example:
  - How effective is it to target the most popular projects? (Section 7.2.1)
  - What sort of benefit would there be from the repository signing rarely updated projects? Which projects should be considered rarely updated? (Section 7.2.2)
  - What is the effect of requiring developers to claim projects when uploading a package? (Section 7.2.3)
  - Is there an effective way to combine these strategies? (Section 7.3)

Quantitative data used to answer these questions was generated using anonymized request logs from PyPI from March 21st to April 19th, 2014. For the purposes of our analysis, we consider a user to be *vulnerable* if the user downloads at least one package that an attacker who compromised the repository (and thus all online keys) could have tampered with. Thus, mapping requests to user devices is important for our analysis. We sanitized the log to remove situations where a single IP address had diverse agent strings (likely multiple systems behind a NAT), or requested the same package more than once (likely a script), or requested more than 100 packages (likely a mirror). Sanitizing the request log reduced the absolute number of IPs in our dataset by about one third to 398,983 users, but provides a data set where each IP very likely corresponds to a unique client. This allows us to take a set of vulnerable packages and understand roughly what percent of clients would request at least one package in that set (and thus would be at risk).

For the purpose of our analysis, we assume that attackers have compromised PyPI on March 21st, at the beginning of the anonymized request log. Furthermore, when analyzing the maximum security model, we assume that all projects that existed before March 21st are delegated by the `claimed-projects` role, and that all projects created afterward during the compromise are delegated by the `new-projects` role.

7.1 Security of Diplomat vs TLS and GPG

We first perform a comparative analysis of the amount of risk placed on users if an attacker compromises (a) a repository protected with TLS, (b) GPG, (c) Diplomat's maximum security model, and (d) Diplomat's legacy security model. This analysis aims to find how effective these solutions would be in practice.

Figure 6 compares the effectiveness of TLS and GPG (the top-most line), and the maximum security model (very near the x-axis). In the case of TLS, the repository is trusted to indicate which packages are valid. Thus, if the repository is compromised, every user is vulnerable because users trust packages from the repository.

Even if GPG is used in conjunction with TLS, the security is not improved. So few developers sign packages and so few users download signatures that there was not a single user who downloaded only GPG-signed packages and their signatures.

Diplomat's maximum security model is not perfect, but it does protect 99.33% of users even if the repository compromise is not detected over the full month's trace. This is because most users only download packages from projects that existed before the start of the trace and those packages are not vulnerable. Users who are vulnerable are those who download a project that was registered during the compromise. This can happen when a new project rapidly becomes popular — often because of the

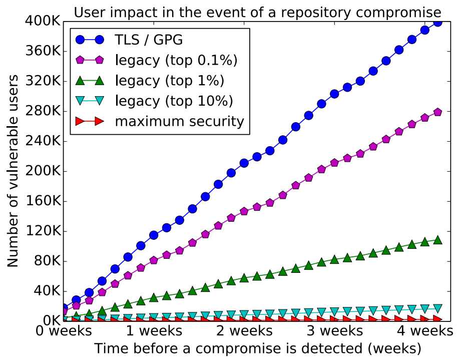


Figure 6: The cumulative number of compromised users over the month when popular projects are signed by developers (lower is better).

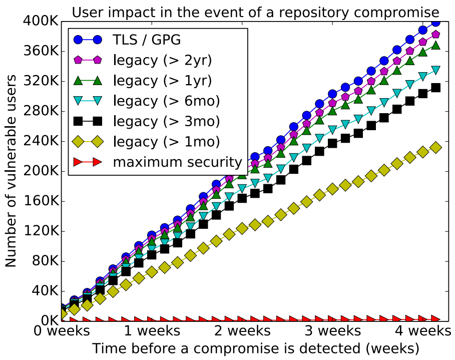


Figure 7: The cumulative number of compromised users over the month when the `rarely-updated-projects` role signed projects that were last updated before the specified time period.

"Slashdot effect" due to promotion on a news site.

7.2 Adoption Strategies in the Legacy Security Model

The compromise-resilience offered by the legacy security model can range from the same as TLS and GPG — none, if no project signs its packages — to as good as the maximum security model, if administrators delegate with offline keys all but new projects to their respective developers. In the rest of this subsection, we explore how the compromise-resilience of the legacy security model differs when different types of projects adopt Diplomat.

7.2.1 Targeting popular projects

We first evaluated the impact of requiring developers of the most popular projects to claim their packages (Figure 6). Increasing the number of popular packages that are signed by developers dramatically increases security. If the most popular 1% of projects are signed by developers (406 projects), then 73% of users are protected. If the top 10% of projects sign their project, then 96% of users are protected. This shows that users overwhelmingly download only popular projects and so focusing on their protection is highly effective.

7.2.2 Only signing rarely updated projects

We examined the security benefits of the repository using an offline key to sign rarely updated projects, because this has very little usability impact until the project is next updated.

As Figure 7 shows, the security benefit of signing rarely updated projects is small. This is because many popular projects are updated frequently. Even if projects that have not updated merely in the last month are considered rarely updated, only 167,097 (42%) of users would be protected if the repository were compromised.

Creating a new package for a rarely updated project means that users will not see the update until repository administrators sign the package with the `rarely-updated-projects` role. This is a major usability problem and so the rate of projects that are considered rarely updated must be very low. To estimate this, we examined the distribution of the maximum time difference between consecutive package updates for all projects (not shown). Our analysis shows that 12% of all packages had a gap of at least a year between updates, but only 4% had a gap of at least two years. We feel that two years is the most aggressive setting for rarely updated projects that is likely to be considered acceptable by the Python community. However, due to the high rate of false positives and low number of users protected, on its own, this is not an effective strategy.

7.2.3 Requiring projects to sign to upload a package

We considered a strategy wherein PyPI would require projects to sign packages in order to upload a new package. (We consider this from a security perspective and ignore the community's response to such a policy.) Figure 8 shows the relative impact on users to be dependent on how long the policy has been in place. For example, the magenta line ("legacy (last 3mo)") shows that if developers that updated a package within the last three months signed that package, 247,969 users (62%) were vulnerable.

The usefulness of requiring a signature to upload a package tails off rather sharply. Somewhat surprisingly, 23% of users (90,091) were vulnerable even if all developers that uploaded a package in the last two years signed it (27,235 projects). This means that many popular projects have not been recently updated. (Given the observation in the previous subsection, many popular packages are updated frequently, yet many are not.) In comparison, this is about as effective as signing the most popular 1% of projects, despite only requiring an action by 406 projects. Thus, requiring projects to sign when uploading is not an effective strategy when used in isolation.

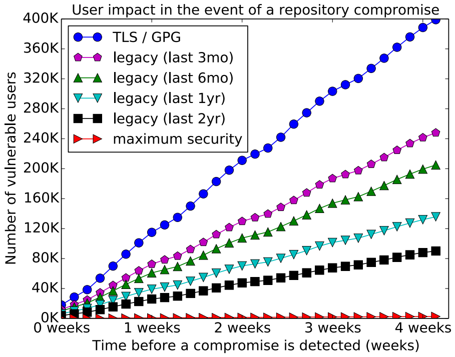


Figure 8: The cumulative number of compromised users over the month when projects were gradually signed by developers over time using the legacy security model.

7.3 Summary: Recommended strategy

To summarize, pushing for adoption by the most popular 1% projects is critical to securing users. Other strategies, such as signing for rarely updated projects and pushing projects to sign when uploading a package, will further help security. While each strategy may be relatively ineffective on its own, combining all of these strategies can have minimal usability impact while greatly increasing the security of PyPI users until the maximum security model is adopted. The details of this analysis, as well as our implementation of Diplomat, are available in our technical report [51]. The Diplomat source code and standards documents are freely available at `https://theupdateframework.github.io/`.

8 Related Work

Role-Based Access Control. Diplomat uses a role-based access control (RBAC) [72, 73] model. RBAC is a collection of security models where permissions are associated with roles. A user may belong to one or more role and thus possess the permissions those roles provide. Although delegations in RBAC are well studied [8–10], ambiguous delegations are not generally studied because users, permissions, and roles are usually implicitly assumed not to conflict. Schaad [74] used Prolog to detect conflicts in separation of duties between roles, but discussed no resolution mechanism that is applicable beyond RBAC. Stork [20] and OrBAC [14, 29] support delegation models that allow the simultaneous permission and prohibition of a privilege, and so could have the same ambiguity issues as this work. These systems solve the problem by specifying *priorities* with every permission and prohibition. However, unlike Diplomat, these systems assume that the metadata a party sees cannot be controlled by the attacker. An attacker that compromises a community repository can choose to omit, add, or, in the case of online keys, alter metadata, which is not handled by these schemes.

Trust Management. Although GPG [83] and X.509 [26] are useful for finding public keys and telling whether keys have been revoked, they cannot answer the question [17]: "Is request *r* authorized by policy *P* and credential set *C*?" In a seminal work [16], Blaze et al. defined *trust management* and introduced PolicyMaker, a general trust management engine to enforce security policies for diverse applications. PolicyMaker separates secure key distribution (as solved by GPG or X.509) from distributed authorization. PolicyMaker featured security principles shared by Diplomat such as *k*-of-*n* thresholds for authorization, deferred trust (delegations), local policies, and key revocation. Later, KeyNote [15] more directly supported public-key infrastructure-like applications with a simpler syntax and semantics at the expense of generality. Unlike Diplomat, PolicyMaker and KeyNote do not handle conflicting statements made by apparently equally trustworthy parties [57]. Like KeyNote, SPKI/SDSI [34] supports trust management for public key infrastructures (PKI). SPKI/SDSI chose a delegation model with boolean control; unlike Diplomat, a key holder can specify the inability to delegate further. SPKI/SDSI considered many security problems that are relevant to Diplomat: key revocation, the risks of online keys and increased key lifetimes, redundancy with a threshold of algorithms or keys, and applying redundancy to replace root keys. However, unlike Diplomat, SPKI/SDSI leaves the processing of delegations, including conflict resolution, to application developers.

Delegation Logic. Diplomat leverages ideas from prior work in logic-based distributed authorization. Much of Diplomat's functionality may be expressed in D2LP, an authorization language in delegation logic [44, 56–58]. D2LP extends early works on trust management and authorization in distributed systems [1, 52] by defining a non-monotonic logic that resolves conflicting conclusions in security policies. D2LP defines a more general notion of prioritized delegations than in Diplomat. For example, although both allow delegators to constrain delegations, D2LP allows delegators to specify arbitrary delegation depth, whereas delegations are always infinitely deep in Diplomat. Furthermore, D2LP allows for partial ordering of rules, but Diplomat requires all delegations to be totally ordered. This means that there will *always* be one trusted conclusion for a package's metadata, or none at all in case no administrator or developer has signed for the package.

Secure Software Updates. Problems with software update security were examined by Bellissimo et al. [13] and Cappos et al. [21]. More recently, Knockel et al. [46] observed that man-in-the-middle attacks on third-party software continue to beleaguer open infrastructure.

The Stork package manager [20, 22, 23], whose security model is also used by popular Linux package managers, addresses a wide array of attacks that involve malicious mirrors. However, this security model assumes that the repository is trustworthy. TUF [71] is designed to securely handle situations where some or all of a repository is compromised. Diplomat leverages the techniques in TUF to protect against certain types of attacks, such as attacks that make valid but outdated packages appear current. However, as was discussed in Section 4, due to the need for online project registration TUF cannot protect a community repository against the most impactful attacks, such providing arbitrarily modified packages.

Revere [55] uses a self-organizing, peer-to-peer overlay network to deliver security updates. It is designed to maximize delivery speed, scalability, high dissemination assurance, and security. In Revere, each peer independently decides on trust relationships with other peers in the overlay network. Thus from a security standpoint, Revere functions somewhat like GPG. In contrast, Diplomat works with a central community repository and delegates trust to projects which do not host content themselves.

Meteor [11] is designed to secure smartphones against multi-market environments. Relevant to Diplomat is their assumption that updates can be malicious due to a compromise of developer keys. They propose independent databases of metadata (such as information about developers, or rating of application binaries by experts) that users consult to determine whether an application should be trusted. Baton [12] showed how smartphone application developers can securely transfer the signing authority of an application to a new developer key without requiring user intervention and a PKI.

Alhamed et al. [2, 3] studied a different approach to securing community repositories. They propose a volunteer community of independent testers who build binaries from releases, certify that binaries come from trusted sources, and attach warnings or even praises to binaries.

Secure Software Repository. The Secure Untrusted Data Repository (SUNDR) [54] addresses a different threat model from Diplomat. SUNDR assumes that the software repository itself cannot be trusted with storing packages. Therefore, each developer checks and signs the history of all file system operations. Developers compare histories with each other in order to ensure *fork consistency*, where developers can eventually detect equivocation. SUNDR requires developers and users to mount a SUNDR file system hosted on the repository, and use its protocol to verify the file system history.

9 Conclusion

This paper presents an architecture that uses prioritized and terminating delegations to secure community repositories. The architecture demonstrates that it is possible to have compromise-resilience in a community repository without sacrificing a defining feature: immediate project registration.

Our system, Diplomat, is flexible enough to enable community repositories to implement different security policies and gracefully transition between them. A community repository can begin with the legacy security model, which provides sufficiently strong protection, but does not require any action by developers. The maximum security model does require that developers sign their packages (or else, new packages cannot be immediately released); however, the security gains are substantial. Diplomat's maximum security model would protect over 99% of PyPI users, even if an attacker controlled the repository undetected for a month.

Acknowledgements

We thank our shepherd, Ramakrishna Kotla, as well as Jon Howell and the anonymous reviewers for their valuable comments. We would also like to thank Lois Anne DeLong and Linda Vigdor for their efforts on this paper, as well as the Docker, Flynn, Haskell, LEAP, OCaml, Python, Ruby, and Square communities for their collaboration. Our work on Diplomat was supported by U.S. National Science Foundation grants CNS-1345049 and CNS-0959138.

References

[1] Abadi, M., Burrows, M., Lampson, B., and Plotkin, G. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst. 15*, 4 (Sept. 1993), 706–734.

[2] Alhamed, K., Silaghi, M. C., Hussien, I., Stansifer, R., and Yang, Y. "Stacking the Deck" Attack on Software Updates: Solution by Distributed Recommendation of Testers. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2013 IEEE/WIC/ACM International Joint Conferences on* (2013), vol. 2, pp. 293–300.

[3] Alhamed, K., Silaghi, M. C., Hussien, I., and Yang, Y. Security by Decentralized Certification of Automatic-Updates for Open Source Software controlled by Volunteers. In *Workshop on Decentralized Coordination* (2013).

[4] Apache Infrastructure Team. apache.org incident report for 8/28/2009. `https://blogs.apache.org/infra/entry/apache_org_downtime_report`, 2009.

[5] Apache Infrastructure Team. apache.org incident report for 04/09/2010. `https://blogs.apache.org/infra/entry/apache_org_04_09_2010`, 2010.

[6] Arcieri, T. Let's figure out a way to start signing RubyGems. `http://tonyarcieri.com/lets-figure-out-a-way-to-start-signing-rubygems`, 2014.

[7] Arkin, B. Adobe to Revoke Code Signing Certificate. `https://blogs.adobe.com/conversations/2012/09/adobe-to-revoke-code-signing-certificate.html`, 2012.

[8] Barka, E., and Sandhu, R. Role-based delegation model/hierarchical roles (RBDM1). In *Computer Security Applications Conference, 2004. 20th Annual* (2004), IEEE, pp. 396–404.

[9] Barka, E., and Sandhu, R. Framework for agent-based role delegation. In *Communications, 2007. ICC'07. IEEE International Conference on* (2007), IEEE, pp. 1361–1367.

[10] Barka, E., Sandhu, R., et al. A role-based delegation model and some extensions. In *23rd National Information Systems Security Conference* (2000), Citeseer, pp. 396–404.

[11] Barrera, D., Enck, W., and van Oorschot, P. C. Meteor: Seeding a security-enhancing infrastructure for multi-market application ecosystems. *IEEE Mobile Security Technologies* (2012).

[12] Barrera, D., McCarney, D., Clark, J., and van Oorschot, P. C. Baton: Key Agility for Android without a Centralized Certificate Infrastructure. Tech. Rep. TR-13-03, School of Computer Science, Carleton University.

[13] Bellissimo, A., Burgess, J., and Fu, K. Secure software updates: disappointments and new challenges. *Proceedings of USENIX Hot Topics in Security (HotSec)* (2006).

[14] Ben-Ghorbel-Talbi, M., Cuppens, F., Cuppens-Boulahia, N., and Bouhoula, A. A delegation model for extended RBAC. *International journal of information security 9*, 3 (2010), 209–236.

[15] Blaze, M., Feigenbaum, J., and Keromytis, A. D. Keynote: Trust management for public-key infrastructures. In *Security Protocols* (1999), Springer, pp. 59–63.

[16] Blaze, M., Feigenbaum, J., and Lacy, J. Decentralized trust management. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on* (1996), IEEE, pp. 164–173.

[17] Blaze, M., Feigenbaum, J., and Strauss, M. *Financial Cryptography: Second International Conference, FC '98 Anguilla, British West Indies February 23–25, 1998 Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, ch. Compliance checking in the PolicyMaker trust management system, pp. 254–274.

[18] Bratko, I. *Prolog programming for artificial intelligence*. Pearson education, 2001.

[19] Brown, M., and Dynamic Network Services, Inc. Pakistan hijacks YouTube. http://research.dyn.com/2008/02/pakistan-hijacks-youtube-1/, 2008.

[20] Cappos, J., Baker, S., Plichta, J., Nyugen, D., Hardies, J., Borgard, M., Johnston, J., and Hartman, J. H. Stork: package management for distributed VM environments. In *The 21st Large Installation System Administration Conference, LISA'07* (2007).

[21] Cappos, J., Samuel, J., Baker, S., and Hartman, J. H. A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 565–574.

[22] Cappos, J., Samuel, J., Baker, S., and Hartman, J. H. Package management security. *University of Arizona Technical Report* (2008), 08–02.

[23] Capppos, J. *Stork: Secure Package Management for VM Environments*. Dissertation, University of Arizona, 2008.

[24] Clark, E. [tor-talk] Another fake key for my email address. https://lists.torproject.org/pipermail/tor-talk/2014-March/032308.html, 2014.

[25] CloudFlare, Inc. Answering the Critical Question: Can You Get Private SSL Keys Using Heartbleed? https://blog.cloudflare.com/answering-the-critical-question-can-you-get-private-ssl-keys-using-heartbleed/, 2014.

[26] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. *The Internet Society* (2008). https://tools.ietf.org/html/rfc5280.

[27] Corbet, J. An attempt to backdoor the kernel. http://lwn.net/Articles/57135/, 2003.

[28] Corbet, J. The cracking of kernel.org. http://www.linuxfoundation.org/news-media/blogs/browse/2011/08/cracking-kernelorg, 2011.

[29] Cuppens, F., Cuppens-Boulahia, N., and Ghorbel, M. B. High level conflict management strategies in advanced access control models. *Electronic Notes in Theoretical Computer Science 186* (2007), 3–26.

[30] Debian. Debian Investigation Report after Server Compromises. https://www.debian.org/News/2003/20031202, 2003.

[31] Debian. Security breach on the Debian wiki 2012-07-25. https://wiki.debian.org/DebianWiki/SecurityIncident2012, 2012.

[32] Docker Inc. Docker Hub. https://hub.docker.com/.

[33] Eklektix, Inc. Docker image "verification". https://lwn.net/Articles/628343/, 2015.

[34] Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., and Ylonen, T. RFC 2693: SPKI certificate theory. https://tools.ietf.org/html/rfc2693.

[35] Frields, P. W. Infrastructure report, 2008-08-22 UTC 1200. https://www.redhat.com/archives/fedora-announce-list/2008-August/msg00012.html, 2008.

[36] Gentoo Linux. rsync.gentoo.org: rotation server compromised. https://security.gentoo.org/glsa/200312-01, 2003.

[37] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 38–49.

[38] Gesbert, L., and Mehnert, H. Signing the OPAM repository. http://opam.ocaml.org/blog/Signing-the-opam-repository/, 2015.

[39] GitHub, Inc. Public Key Security Vulnerability and Mitigation. https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation, 2012.

[40] GNU Savannah. Compromise2010. https://savannah.gnu.org/maintenance/Compromise2010/, 2010.

[41] Goodin, D. Attackers sign malware using crypto certificate stolen from Opera Software. http://arstechnica.com/security/2013/06/attackers-sign-malware-using-crypto-certificate-stolen-from-opera-software/, 2013.

[42] Google, Inc. Open Client Update Protocol. http://omaha.googlecode.com/svn/wiki/cup.html.

[43] Gostev, A. 'Gadget' in the middle: Flame malware spreading vector identified. https://www.securelist.com/en/blog/208193558/Gadget_in_the_middle_Flame_malware_spreading_vector_identified, 2012.

[44] Grosof, B. N. Prioritized conflict handling for logic programs. In *ILPS* (1997), vol. 97, pp. 197–211.

[45] Inci, M. S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., and Sunar, B. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. Cryptology ePrint Archive, Report 2015/898, 2015. http://eprint.iacr.org/.

[46] Knockel, J., and Crandall, J. R. Protecting Free and Open Communications on the Internet Against Man-in-the-Middle Attacks on Third-Party Software: We're FOCI'd. In *Presented as part of the 2nd USENIX Workshop on Free and Open Communications on the Internet* (Berkeley, CA, 2012), USENIX.

[47] Krah, S. [Python-Dev] pip: cdecimal an externally hosted file and may be unreliable [sic]. `https://mail.python.org/pipermail/python-dev/2014-May/134453.html`, 2014.

[48] Kuhn, B. M. News: IMPORTANT: Information Regarding Savannah Restoration for All Users. `https://savannah.gnu.org/forum/forum.php?forum_id=2752`, 2003.

[49] Kuppusamy, T. K., Diaz, V., Stufft, D., and Cappos, J. PEP 458 – Securing the Link from PyPI to the End User. `https://www.python.org/dev/peps/pep-0458/`, 2013.

[50] Kuppusamy, T. K., Diaz, V., Stufft, D., and Cappos, J. PEP 480 – Surviving a Compromise of PyPI. `https://www.python.org/dev/peps/pep-0480/`, 2014.

[51] Kuppusamy, T. K., Torres-Arias, S., Diaz, V., and Cappos, J. Diplomat: Using Delegations to Protect Community Repositories. Tech. Rep. TR-CSE-2016-01, Computer Science and Engineering, Tandon School of Engineering, New York University.

[52] Lampson, B., Abadi, M., Burrows, M., and Wobber, E. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 265–310.

[53] LEAP Encryption Access Project. New releases for a new year - LEAP. `https://leap.se/en/2014/darkest-night`, 2014.

[54] Li, J., Krohn, M., Mazières, D., and Shasha, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 9–9.

[55] Li, J., Reiher, P., and Popek, G. J. Resilient self-organizing overlay networks for security update delivery. *Selected Areas in Communications, IEEE Journal on 22*, 1 (2004), 189–202.

[56] Li, N. *Delegation Logic: A Logic-based Approach to Distributed Authorization*. PhD thesis, New York University, 2000.

[57] Li, N., Feigenbaum, J., and Grosof, B. N. A logic-based knowledge representation for authorization with delegation. In *Computer Security Foundations Workshop, 1999. Proceedings of the 12th IEEE* (1999), IEEE, pp. 162–174.

[58] Li, N., Grosof, B. N., and Feigenbaum, J. A Nonmonotonic Delegation Logic with Prioritized Conflict Handling. `https://www.cs.purdue.edu/homes/ninghui/papers/old/d2lp.pdf`, 2000.

[59] Magnusson, H. The PHP project and Code Review. `http://bjori.blogspot.com/2010/12/php-project-and-code-review.html`, 2010.

[60] Microsoft, Inc. Order of Windows Firewall with Advanced Security Rules Evaluation. `https://technet.microsoft.com/en-us/library/cc755191%28v=ws.10%29.aspx`, 2009.

[61] Microsoft, Inc. Flame malware collision attack explained. `http://blogs.technet.com/b/srd/archive/2012/06/06/more-information-about-the-digital-certificates-used-to-sign-the-flame-malware.aspx`, 2012.

[62] Mullenweg, M. Passwords Reset. `https://wordpress.org/news/2011/06/passwords-reset/`, 2011.

[63] Mónica, D., and Docker, Inc. Introducing Docker Content Trust. `https://blog.docker.com/2015/08/content-trust-docker-1-8/`, 2015.

[64] Philips, B. Evaluate The Update Framework. `https://github.com/appc/spec/issues/211`, 2015.

[65] Prime Directive, Inc. Development - Flynn. `https://flynn.io/docs/development`, 2015.

[66] Python Software Foundation. PyPI - the Python Package Index: Python Package Index. `https://pypi.python.org/pypi`.

[67] Red Hat, Inc. Infrastructure report, 2008-08-22 UTC 1200. `https://rhn.redhat.com/errata/RHSA-2008-0855.html`, 2008.

[68] RubyGems.org. RubyGems.org — your community gem host. `https://rubygems.org/`.

[69] RubyGems.org. Security. `http://guides.rubygems.org/security/`.

[70] RubyGems.org. Data Verification. `http://blog.rubygems.org/2013/01/31/data-verification.html`, 2013.

[71] Samuel, J., Mathewson, N., Cappos, J., and Dingledine, R. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 61–72.

[72] Sandhu, R. S. Role-based access control. *Advances in computers 46* (1998), 237–286.

[73] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. Role-based access control models. *Computer 29*, 2 (1996), 38–47.

[74] Schaad, A. Detecting conflicts in a role-based delegation model. In *Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual* (2001), IEEE, pp. 117–126.

[75] Shay, X., and Square, Inc. Securing RubyGems with TUF, Part 1. `https://corner.squareup.com/2013/12/securing-rubygems-with-tuf-part-1.html`, 2013.

[76] Shay, X., and Square, Inc. Securing RubyGems with TUF, Part 2. `https://corner.squareup.com/2013/12/securing-rubygems-with-tuf-part-2.html`, 2013.

[77] Shay, X., and Square, Inc. Securing RubyGems with TUF, Part 3. `https://corner.squareup.com/2013/12/securing-rubygems-with-tuf-part-3.html`, 2013.

[78] Slashdot Media. About. `http://sourceforge.net/about`.

[79] Slashdot Media. phpMyAdmin corrupted copy on Korean mirror server. `https://sourceforge.net/blog/phpmyadmin-back-door/`, 2012.

[80] Smith, J. K. Security incident on Fedora infrastructure on 23 Jan 2011. `https://lists.fedoraproject.org/pipermail/announce/2011-January/002911.html`, 2011.

[81] Stewart, J. DNS cache poisoning–the next generation. `http://www.secureworks.com/research/articles/dns-cache-poisoning`, 2003.

[82] The FreeBSD Project. FreeBSD.org intrusion announced November 17th 2012. `http://www.freebsd.org/news/2012-compromise.html`, 2012.

[83] The GnuPG Project. The GNU Privacy Guard. `https://gnupg.org/`.

[84] The MITRE Corporation. CVE 2008-0166. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166`, 2008.

[85] The MITRE Corporation. CVE 2014-0092. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0092`, 2014.

[86] The PHP Group. php.net security notice. `http://www.php.net/archive/2011.php#id2011-03-19-1`, 2011.

[87] The PHP Group. A further update on php.net. `http://php.net/archive/2013.php#id2013-10-24-2`, 2013.

[88] THE UPDATE FRAMEWORK. Developer Tools. https://github.com/theupdateframework/tuf/blob/develop/tuf/README-developer-tools.md.

[89] THE UPDATE FRAMEWORK. Repository Management. https://github.com/theupdateframework/tuf/blob/develop/tuf/README.md.

[90] VOSS, L. Newly Paranoid Maintainers. http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers, 2014.

[91] WELL-TYPED LLP. Improving Hackage security. http://www.well-typed.com/blog/2015/04/improving-hackage-security/, 2015.

[92] WOBBER, E., ABADI, M., BURROWS, M., AND LAMPSON, B. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS) 12*, 1 (1994), 3–32.

AnonRep: Towards Tracking-Resistant Anonymous Reputation

Ennan Zhai[†] David Isaac Wolinsky[‡] Ruichuan Chen[§]
Ewa Syta[†] Chao Teng[‡] Bryan Ford[*]
[†]Yale University [‡]Facebook Inc. [§]Nokia Bell Labs [*]EPFL

Abstract

Reputation systems help users evaluate information quality and incentivize civilized behavior, often by tallying feedback from other users such as "likes" or votes and linking these scores to a user's long-term identity. This identity linkage enables user tracking, however, and appears at odds with strong privacy or anonymity. This paper presents AnonRep, a practical anonymous reputation system offering the benefits of reputation without enabling long-term tracking. AnonRep users anonymously post messages, which they can verifiably tag with their reputation scores without leaking sensitive information. AnonRep reliably tallies other users' feedback (*e.g.*, likes or votes) without revealing the user's identity or exact score to anyone, while maintaining security against score tampering or duplicate feedback. A working prototype demonstrates that AnonRep scales linearly with the number of participating users. Experiments show that the latency for a user to generate anonymous feedback is less than ten seconds in a 10,000-user anonymity group.

1 Introduction

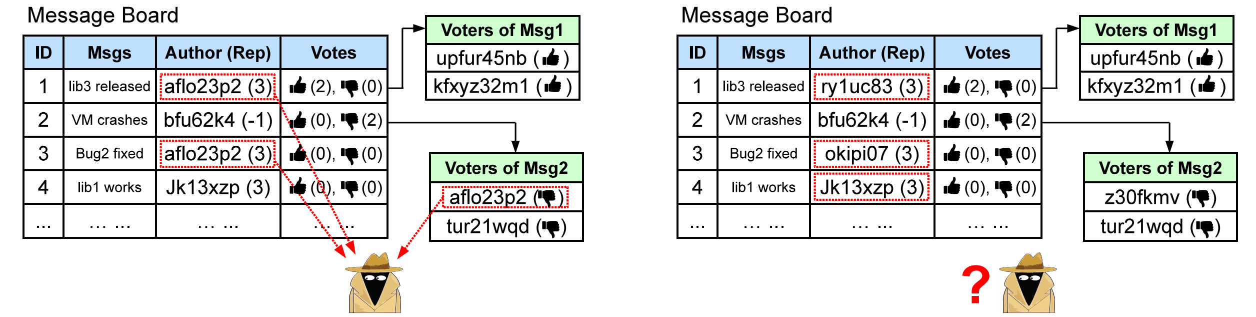
Online services such as eBay, Yelp, and Stack Overflow employ reputation systems to evaluate information quality and filter spam. In Yelp, for example, users post messages (*e.g.*, reviews), and offer feedback on other users' posts (*e.g.*, votes) based on perceived utility. User reputations increase or decrease based on this feedback, and reputation affects how widely a user's future posts are viewed. This long-term linkage between user behavior and reputation, however, can quickly de-anonymize users wishing to hide their true identities [4,23,28,31]. For example, Minkus *et al.* [28] revealed eBay users' sensitive purchase histories by analyzing only pseudonyms' transactions and feedback. As privacy has become a major concern for online users, we raise the question: can we combine the benefits of reputation with the privacy afforded by fully anonymous, unlinkable messaging? Can we build an *anonymous reputation system*?

In an anonymous reputation system, no entity – either users or servers implementing the reputation system – should be able to link posted messages and feedback to any user identity. Maintaining reputation without identity in principle offers the benefits of reputation with unprecedented user privacy [5]. Achieving this goal is challenging, however, since the requirement to associate users with their historical activities seems to preclude anonymity [38]. Establishing reputation while maintaining unlinkability of activities appears to be a paradox.

Prior efforts have addressed this problem [2, 5, 7, 15], but none have yet proven practical or sufficiently general for realistic deployment. For example, Androulaki *et al.* [2] proposed blind signatures for anonymous peer-to-peer reputation transactions, but this protocol relies on a centralized honest entity and cannot support negative feedback (*e.g.*, about trolls or otherwise misbehaving users). Similarly, Bethencourt *et al.* [5] proposed to use zero-knowledge proofs to construct signatures of reputation, thus keeping unlinkability of users' historical behaviors. This approach supports limited reputation algorithms, however, and is computationally expensive.

This paper presents *AnonRep*, the first practical anonymous reputation system supporting diverse reputation schemes without leaking sensitive information about users' long-term identities or historical activities. AnonRep represents a novel integration of known cryptographic primitives – verifiable shuffles [32], linkable ring signatures [26], and homomorphic crypto [17] – in a multi-provider deployment architecture. AnonRep builds on the anytrust model [41], like Dissent [42] and Vuvuzela [39], for scalability and robustness to client churn. An AnonRep group consists of a potentially large set of client nodes representing users, and a smaller set of third-party commodity servers implementing the anonymous reputation service. Each client trusts that at least one server is honest and not colluding with the others, but the client need not know which server to trust.

AnonRep operates in a series of *message-and-feedback* rounds. Each round might in practice last a few minutes, hours, or even a full day, depending on the application scenario. At the beginning of each round, the servers maintain a database containing all clients' long-term identities and their respective encrypted reputation scores. During each round, the servers successively run a scheduling protocol based on verifiable shuffles [32], which transforms the reputation list into an anonymously permuted list consisting of a one-time pseudonym for each client and an associated plaintext reputation score. While exact reputation scores could themselves link clients across rounds, AnonRep allows users to reveal only approximate reputations (§6). AnonRep's scheduling protocol is decentralized: neither servers nor clients



Message Board

ID	Msgs	Author (Rep)	Votes
1	lib3 released	aflo23p2 (3)	👍 (2), 👎 (0)
2	VM crashes	bfu62k4 (-1)	👍 (0), 👎 (2)
3	Bug2 fixed	aflo23p2 (3)	👍 (0), 👎 (0)
4	lib1 works	Jk13xzp (3)	👍 (0), 👎 (0)
...	... ...	... ...	... ...

Voters of Msg1
upfur45nb (👍)
kfxyz32m1 (👍)

Voters of Msg2
aflo23p2 (👎)
tur21wqd (👎)

(a) A typical message board equipped with conventional reputation technique which potentially has linkability issues.

Message Board

ID	Msgs	Author (Rep)	Votes
1	lib3 released	ry1uc83 (3)	👍 (2), 👎 (0)
2	VM crashes	bfu62k4 (-1)	👍 (0), 👎 (2)
3	Bug2 fixed	okipi07 (3)	👍 (0), 👎 (0)
4	lib1 works	Jk13xzp (3)	👍 (0), 👎 (0)
...	... ...	... ...	... ...

Voters of Msg1
upfur45nb (👍)
kfxyz32m1 (👍)

Voters of Msg2
z30fkmv (👎)
tur21wqd (👎)

(b) The same message board equipped with AnonRep. Adversary cannot link different activities to any specific identity.

Figure 1: Motivating example. We use a typical message board with different reputation technique. The sample message board evaluates the quality of posted messages based on the reputation of these messages' authors. Feedback is represented as votes.

(other than the owner) can link one-time pseudonyms or reputations to long-term identities.

Clients then post messages anonymously using these one-time pseudonyms. The servers can associate these messages with their corresponding reputation scores without learning clients' sensitive information. Each client may then provide feedback (*e.g.*, votes) on other clients' posted messages. Each vote is signed by a linkable ring signature [26], enabling the servers to verify that each client votes only once without revealing which client submitted each vote. This design enables the servers to tally positive and negative feedback without linking this feedback with long-term identities.

Finally, the servers tally the feedback received for each one-time pseudonym, update the reputation score, and then perform a "reverse scheduling" to transform these one-time pseudonyms and their updated reputation scores back to the original long-term identities and their encrypted updated reputation scores.

We have implemented an AnonRep prototype in Go. Experimental results show that the AnonRep server scales linearly with the number of clients. With a 10,000-client anonymity group, for example, each server's computational cost is about one minute per round. The time required for a client to construct an anonymous vote to provide feedback is less than ten seconds in a 10,000-client anonymity group. While the current prototype has many limitations and would benefit from further development, we nevertheless believe that AnonRep represents a significant step towards building a practical anonymous reputation system for realistic online services.

In summary, this paper makes the following contributions. First, we propose the first practical anonymous reputation system, AnonRep, offering the benefits of reputation while maintaining the unlinkability and anonymity of users' historical activities. Second, we provide a fully-functional open-source prototype illustrating AnonRep's functionality and practicality.

2 Motivation and Challenges

This section first presents a simple but illustrative example to motivate AnonRep's goals (§2.1), then discusses key technical challenges (§2.2).

2.1 Motivating Example

Figure 1a shows a typical reputation system, which utilizes a message board to evaluate information quality and filter out spam. The message board maintains a reputation score for each client. Each client has an identity or pseudonym, which remains fixed throughout the client's lifetime. Suppose a client, Alice, posts a message on the message board. The message board associates the message with Alice's reputation score, which other users or content curation algorithms might use to determine how widely Alice's message is seen. Other clients who view Alice's message can then give positive or negative feedback to express subjective opinions on the quality of Alice's message. Based on this feedback, the message board updates Alice's reputation, enabling Alice to post new messages with the updated reputation. Precisely how user feedback affects clients' reputation scores varies depending on the specific reputation algorithm.

Message board reputation systems of this kind have been widely employed by many online services, *e.g.*, eBay, Yelp, and Stack Overflow. However, in such a system, each client's reputation score is associated with either her real identity or a long-lived pseudonym. As a result, even with pseudonyms, a client's historical activities can be easily tracked and linked, leaking sensitive information. For example, in Figure 1a, an adversary can observe that the first and third messages are posted by the same pseudonym `aflo23p2`, and the second message is posted by another pseudonym dged2p. The adversary can also learn the voters of each message. For example, the client with pseudonym aflo23p2 casts a negative vote to the second message. Even in the absence of clients' real identities, Minkus *et al.* [28] have successfully exposed eBay clients' sensitive purchase histories and feedback by analyzing only pseudonyms' transactions.

The goal of this paper is to design a practical anonymous reputation system providing the utility of a reputation system, while hiding clients' sensitive information – including the linkage between messages posted by the same user. With AnonRep, as shown in Figure 1b, a client appears as different one-time pseudonym every time the client posts a message. These one-time pseudonyms avoid revealing information that can link any clients' messages, reputation scores, or votes across posting rounds. Meanwhile, AnonRep can still privately update each client's reputation score without any participants learning sensitive information.

While content-based attacks such as stylometry [30] could still link one user's message across rounds, these techniques are uncertain and prone to false positives, especially operating on short messages (*e.g.*, tweets). Regardless, AnonRep's goal is not to address content-based linkage risks, but to ensure that feedback and reputation management does not leak any *more* sensitive information beyond what the user-provided content itself might.

2.2 Technical Challenges

We face two main technical challenges to build a practical anonymous reputation system.

Challenge 1: Protecting the association between reputation and identity. The calculation of a user's reputation score relies on the historical activities associated with this user's identity. It seems that maintaining this reputation would preclude any possibility of identity anonymity [2, 5]. Two straightforward solutions are 1) to introduce a trusted third party that updates reputation scores for clients, or 2) to use secure multi-party computation (SMPC) [44] to update reputation scores privately. Unfortunately, the former solution offers weak security by requiring every user to trust a third party, while the latter solution is slow and computation-intensive and has not proven scalable in practice. Therefore, keeping the association between reputation and user identity private presents a significant challenge.

Challenge 2: Detection of misbehavior. A centralized reputation system can readily enforce well-defined rules for handling reputation and feedback fairly – such as "one client, one vote" – because a trusted entity can see all clients' activities and enforce these rules. In a decentralized anonymous reputation system without a trusted party, however, clients might misbehave, *e.g.*, by casting multiple votes on the same message to amplify the user's feedback unfairly. Such misbehavior is non-trivial to detect in an anonymous reputation system [5].

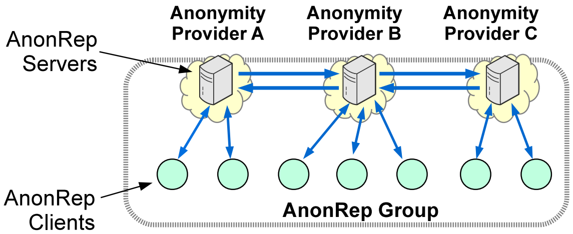


Figure 2: AnonRep's multi-provider deployment model, and its basic communication topology. In an AnonRep group, each client communicates with a single upstream server, while each server can communicate with all other servers.

3 AnonRep Overview

In this section, we first sketch the architecture of AnonRep in §3.1. Then, we present our assumptions and threat model in §3.2. Finally, we give our security goals in §3.3.

3.1 Architecture

As illustrated in Figure 2, AnonRep relies on a *multi-provider* model to achieve scalability and resilience to link failures [16, 41, 42]. A typical AnonRep group includes two types of members: 1) a potentially large number of unreliable *client* nodes representing individual users, and 2) a small number of *servers*, which are assumed to be highly available and well-provisioned.

In practice, each server in an AnonRep group should be operated independently (*i.e.*, each managed by a separate operator) to limit the risks of all servers being compromised or colluding against clients.

Each client directly communicates with at least one *upstream server*, while each server can communicate with any other servers (see Figure 2). Such a communication topology reduces the communication and computational overhead at the clients, and enables the system to tolerate client churn [42]. More specifically, each client does not need to know which other clients are online while posting messages or feedback to the upstream server.

3.2 Threat Model and Assumptions

In an AnonRep group, clients need *not* assume any particular server is trustworthy, and they need not even trust their respective upstream servers. Instead, we assume the anytrust model, *i.e.*, each client trusts only that there exists *at least one* honest server without knowing which this server is [16, 41, 42]. An AnonRep group member (server or client) is *honest* if the member follows the specified protocol faithfully and does not collude with or leak sensitive information to other group members. A member is *dishonest* (or malicious) otherwise.

A malicious client may wish to link or track sensitive information such as reputation scores, messages, and feedback to specific victim clients. Multiple malicious clients may collude with each other.

A malicious server may refuse to service honest clients, but such refusal should not compromise clients' anonymity. Moreover, a malicious server may try to tamper with clients' reputation scores, and even collude with malicious clients.

We assume that public and symmetric key encryptions, key-exchange mechanisms, signature schemes and hash functions are all correctly used. We also assume that public keys of AnonRep servers and clients are publicly available. We assume the network connections between clients and servers are established over anonymous communication channels (*e.g.*, Tor [19], or traffic analysis resistant networks like Dissent [42] and Vuvuzela [39]).

3.3 Security Goals

Anonymity. The main goal of AnonRep is to achieve *anonymity* for its clients in face of a strong adversary, *i.e.*, malicious servers and clients as defined above. In AnonRep, anonymity means not only the privacy of each client's data such as profile and IP location, but also the unlinkability of clients' historical activities. No AnonRep group member should be able to link a specific client's sensitive information such as posted messages, reputation scores, or feedback to the client's identity.

AnonRep provides the above anonymity guarantee among the set of honest group members, that is, members who faithfully follow our protocol. AnonRep does not attempt to provide anonymity to malicious group members as they can may collude with others and reveal their identities and association to their messages themselves.

Other goals. Besides anonymity, AnonRep should ensure that the misbehaviors of malicious group members are detectable. In addition, AnonRep should balance the trade-offs between practicality and security. The more clients an anonymity group contains, the stronger anonymity it can offer but at the cost of higher overhead.

Non-goals. Like many prior reputation systems, AnonRep is not designed against the Sybil attack where attackers generate a large number of fake clients to manipulate the reputation of honest clients. How to make AnonRep resistant to the Sybil attack is out of the scope of this paper. In addition, AnonRep is not resilient to network-level Denial-of-Service (DoS) attacks where attackers, for instance, could target the AnonRep servers. The servers, however, are assumed to be highly available. Nevertheless, some well-known defenses (*e.g.*, server provisioning and proof-of-work challenges) could be deployed to mitigate DoS attacks.

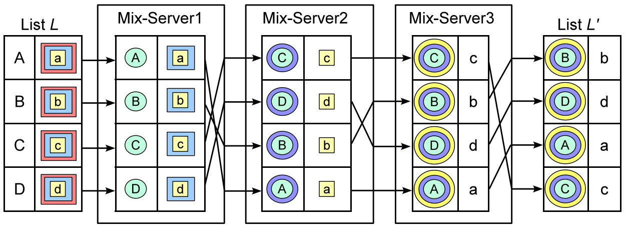


Figure 3: A simple Mix-Net example with three mix-servers. Each mix-server performs the verifiable shuffle protocol. The input is a two-column list with four entries. Each mix-server 1) encrypts and decrypts the elements in the first and second columns with different keys, respectively; 2) permutes the order of entries in the list; and 3) sends the permuted list to the next mix-server.

4 Cryptographic Building Blocks

Before we elaborate on the design details of AnonRep in §5, we first describe several cryptographic techniques that AnonRep builds upon.

4.1 Mix-Net and Verifiable Shuffles

Mix-Net [12] is a decentralized cryptographic protocol that creates hard-to-trace communications by using a chain of servers (called *mixes or mix-servers*) which take in a list of objects, shuffle them, and then output them in random order. As shown in Figure 3, such a primitive ensures unlinkability between the source and the destination of the list.

The shuffle phase in a Mix-Net protocol contains encryption, decryption and permutation operations. For example in Figure 3, each mix-server adds one ciphertext layer on each element in the first column of the received list, strips one ciphertext layer from each element in the second column, then permutes entries in the list, and finally sends the resulting list to the next mix-server.

In order to ensure the correctness of the operations performed in shuffle phases, many verifiable shuffle protocols have been proposed [25, 32, 33]. In a typical verifiable shuffle protocol, besides performing the shuffle operartions, each mix-server generates a zero-knowledge proof, which can be used by any observer (*i.e.*, verifier) to check whether the mix-server correctly performed its shuffle.

Here, we detail a verifiable shuffle primitive `Shuffle`$(g_i, \vec{L}_i, z_i, e_i)$ that we use in the AnonRep design (§5.3). Figure 3 presents an example where three mix-servers successively run this primitive. In a typical `Shuffle` primitive execution, each mix-server i performs the following four operations.

1. Use the public key e_i to encrypt each element in the first column of list $\vec{L}_i$; use the private key z_i to strip one ciphertext layer from each element in the second column of the list $\vec{L}_i$;

2. Permute entries in the resulting list, producing $\vec{L_{i+1}}$;
3. Use the public key e_i to encrypt the received generator g_i, *i.e.*, $g_{i+1} = {g_i}^{e_i} \bmod p$; and
4. Generate a (zero-knowledge) proof f_i attesting that the above operations are correctly performed.

After running the primitive, the mix-server i sends $\vec{L_{i+1}}$, g_{i+1} with the proof f_i to the next mix-server $i+1$.

4.2 Linkable Ring Signatures

Liu *et al.* proposed linkable ring signatures [26], a variant of traditional ring signatures [37]. A linkable ring signature allows any of n group members to produce a ring signature on some message such that no one knows *which* group member produced the signature but all signatures from the same member can be linked together.

Each group member holds a public/private key pair (PK_i, SK_i). Member i can compute a ring signature σ on a message m, on input $(m, SK_i, PK_1, ..., PK_n)$. Anyone can check the validity of a ring signature given (σ, m) and the public key list $L = \{PK_1, ..., PK_n\}$ of all group members; however, nobody knows who signed the message m. It is hard for anyone to create a valid ring signature on any message on behalf of some group without knowing at least one secret key of a member of this group. Another important property of linkable ring signature is *linkability*: given any two signatures, a verifier can determine whether they were produced by the same member in the group but still without learning the specific member's identity.

In particular, linkable ring signature consists of the following four steps [26].

Initialization step: Each member i $(i = 1, ..., n)$ has a public key Y_i, and private key y_i, where $(Y_i = g^{y_i})$. Each member knows the list of n members' public keys $L = \{Y_1, ..., Y_n\}$, and a public hash function $H(\cdot)$.

Signature generation step: Suppose a member i, called a signer, wants to use linkable ring signature scheme to sign a message m. She first needs to compute the linkability tag $t = H(L)$. Then, i runs the primitive `Sign`(m, L, y_i, t) to get m's linkable ring signature $\sigma(m)$. Finally, i sends m and $\sigma(m)$ to the verifier.

Verification step: The verifier receives the message m and the signature $\sigma(m)$. He knows the public key list L. The verifier runs `Verify`(t, m, L, $\sigma(m)$) to check whether $\sigma(m)$ is produced by one of the members in the group specified by L.

Linkability checking step: Given two signatures $\sigma'(m')$ and $\sigma''(m'')$, the verifier can check whether the two signatures are from the same signer by running `Check`($\sigma'(m')$, $\sigma''(m'')$). Because each linkable ring signature is generated based on a linkability tag $t = H(L)$ and the private key of signer y_i, if the two signatures are from the same signer, the verifier would successfully confirm this fact.

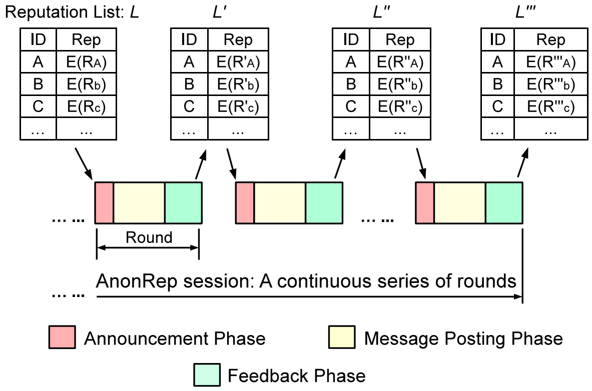


Figure 4: AnonRep's session, rounds and phases. An AnonRep session contains a continuous series of message-and-feedback rounds. Each round has three phases: 1) announcement phase, 2) message posting phase, and 3) feedback phase. All the online (or available) members, including servers and clients, synchronously participate in these rounds.

5 AnonRep System Design

This section details AnonRep's basic design (§5.1-§5.5), followed by practical considerations (§5.6).

5.1 AnonRep Workflow

A typical AnonRep session, as shown in Figure 4, consists of a series of *message-and-feedback* rounds. All online AnonRep group members (including servers and clients) synchronously participate in these rounds. In practice, the duration of each round may be a few hours or even one day, depending on the application scenario.

The input to each round is a two-column *reputation list*. The first column records the long-term identity of each registered client, and the second column is the client's reputation score encrypted by all servers (see §5.2). A client's long-term identity is her public key, which corresponds to a private key maintained by the client herself. The output of each round is a similar reputation list with updated clients' reputation scores. The output list of one round serves as input to the next round, as shown in Figure 4. Any newcomer client can participate in the AnonRep session after she completes the registration process (details in §5.2).

Each round consists of three phases. The duration of each phase may be significantly different.

- **Announcement phase:** Servers run *scheduling protocols* to assign a one-time pseudonym to each client. Only the client knows and can use the one-time pseudonym assigned to her (§5.3).
- **Message posting phase:** Clients anonymously post messages using the assigned one-time pseudonyms, and the upstream servers associate the corresponding reputation scores with the messages, without learning clients' long-term identities (§5.4).

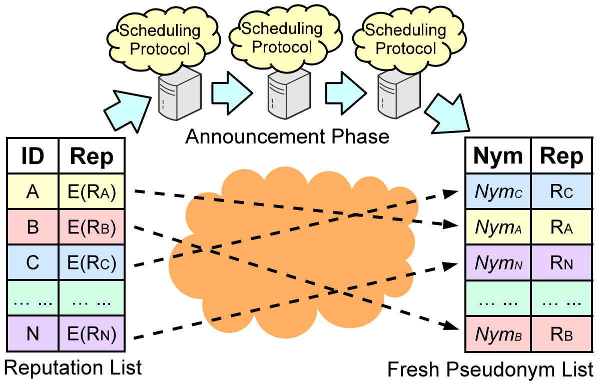


Figure 5: AnonRep's announcement phase via scheduling protocols. Each entry in the reputation list records some client's long-term identity and the ciphertext of her reputation score $E(R_i)$, which has been encrypted by all the servers. On the other side, each entry in the fresh pseudonym list records some client's one-time pseudonym for this round and the plaintext of her reputation score R_i.

- **Feedback phase:** Clients anonymously provide feedback to posted messages. At the end of this phase, servers update each one-time pseudonym's reputation score based on the received feedback, and then update the long-term scores in the reputation list by running "reverse scheduling" protocols (§5.5).

With the system overview in place, we now describe the details of the system design.

5.2 Client Registration

Any newcomer client who wants to use AnonRep needs to register with the servers.

Specifically, each new client i *first* generates a key-pair $\langle Y_i = g^{y_i}, y_i \rangle$, where g is a generator shared among servers and clients. Here, Y_i and y_i are the client i's long-term public and private keys, respectively. *Then*, the client i uploads the public key Y_i to a randomly selected AnonRep server S_j, called the client i's upstream server. *Next*, S_j creates an initial reputation score r_i for the client i, and then encrypts this initial reputation score by using its public key Z_j, and sends the ciphertext to the next server, *i.e.*, S_{j+1}. All servers use their public keys to encrypt this initial reputation in sequence. Once the client i's upstream server S_j receives the client's reputation score $E(r_i)$ which has been encrypted by all servers, S_j creates a tuple $\langle Y_i, E(r_i) \rangle$, and broadcasts this tuple to the servers. *Finally*, each AnonRep server appends this tuple to a local reputation list.

5.3 Announcement Phase

As shown in Figure 4, the announcement phase is the first phase of a message-and-feedback round. In a typical announcement phase, as shown in Figure 5, the servers take the reputation list as input, and successively perform *scheduling protocols* to generate a *fresh pseudonym list*, in order to enable each client to have a "temporary identity" to post message and provide feedback in the following phases. The entries in the fresh pseudonym list are in a permuted order. Each entry corresponds to one client, and contains a one-time pseudonym as well as the plaintext reputation score for this client. Because the announcement phase is executed by multiple independent servers, no server can link the original reputation list to the generated fresh pseudonym list as long as at least one server does not collude with others. Moreover, each client only knows her own entry in the fresh pseudonym list, and cannot learn the associations between other clients and their pseudonyms.

At the beginning of an announcement phase, each server j locally maintains an ephemeral secret e_j (different in each round), a public generator g, and its own private key z_j, which corresponds to a public key, Z_j, used to encrypt the new client's reputation score during the client registration. The servers perform the scheduling protocol (shown in Algorithm 1), transforming the input reputation list $\vec{L}$ and the public generator g into the fresh pseudonym list $\vec{pk}$ and the final generator $g_{m+1} = g^{e_1 \dots e_m}$.

After all the servers finish the scheduling protocol, each client learns the fresh pseudonym list, *i.e.*, $\vec{pk}$, and the final generator, g_{m+1}, from her upstream server. Then, each client i is able to compute and find her own one-time pseudonym, $pk_{\pi(i)}$, in $\vec{pk}$ by: $pk_{\pi(i)} = {g_{m+1}}^{y_i}$, where y_i is the client i's private key (corresponding to her long-term public key Y_i, defined in §5.2), and $\pi(i)$ denotes the location of client i's one-time pseudonym in the fresh pseudonym list $\vec{pk}$.

Based on the working principle of verifiable shuffle primitive (see §4.1 and Algorithm 1), client i's long-term pseudonym key Y_i is encrypted by all the servers, *i.e.*, ${Y_i}^{e_1 \dots e_m} = pk_{\pi(i)}$. Because $Y_i = g^{y_i}$ (see §5.2), we have:

$$pk_{\pi(i)} = {Y_i}^{e_1 \dots e_m} = (g^{y_i})^{e_1 \dots e_m} = (g^{e_1 \dots e_m})^{y_i} = {g_{m+1}}^{y_i}$$

Only the client i learns $pk_{\pi(i)}$, since only i knows her private key y_i. In the current round, each client i is assigned a new public/private key-pair $\langle pk_{\pi(i)}, y_i \rangle$ based on the final generator g_{m+1}: $pk_{\pi(i)} = {g_{m+1}}^{y_i}$. Each client can use this one-time pseudonym to post messages and provide feedback later, without leaking the long-term identity.

The scheduling protocol uses verifiable shuffles [32]. Therefore, during the announcement, each server computes and attaches a zero-knowledge proof of correctness to each "intermediate list" sent to its successive server. This step ensures that if a server misbehaves, it will be detectable by other servers.

Algorithm 1 Scheduling protocol.

All members (including clients and servers) know the reputation list $\vec{L}$ and the public generator g. Each server j knows its private key z_j and an ephemeral secret e_j, and each client i knows her public key $Y_i = g^{y_i}$, where y_i is i's private key.

1. The first server S_1 takes the reputation list $\vec{L_1} = \vec{L}$ as input, and performs the verifiable shuffle primitive, `Shuffle`($g = g_1$, $\vec{L_1}$, z_1, e_1), to obtain outputs: g_2, $\vec{L_2}$ and a proof. Then, S_1 sends $\vec{L_2}$, g_2 with the proof to the next server S_2.
2. Each server S_j ($j = 2,...,m$) successively runs the same verifiable shuffle primitive as S_1. Namely, S_j runs `Shuffle`(g_j, $\vec{L_j}$, z_j, e_j) to obtain g_{j+1}, $\vec{L_{j+1}}$ and a proof about the correctness. Then, S_j sends $\vec{L_{j+1}}$, g_{j+1} with the proof to the next server S_{j+1}.
3. After the final server S_m performs the shuffle primitive, S_m outputs the fresh pseudonym list, $\vec{pk} = \vec{L_{m+1}}$, and the final generator, g_{m+1}, which has been encrypted by all the servers. *Note:* Reputation scores in $\vec{pk}$ are plaintexts now, since all the ciphertext layers have already been decrypted by all the servers. Then, servers distribute all results ($\vec{L_j}, g_j \forall j \in m$ with proofs) to all the other servers.
4. Each server k verifies each $\vec{L_j}$, g_j, and proofs. If they match, server k transmits a signature, sig_k, of $\vec{pk}$ to all other servers.
5. Upon collecting a signature sig_j from every other server j, servers distribute $\vec{pk}$, g_{m+1}, and $sig_j \forall j \in m$ to their clients.

5.4 Message Posting Phase

After the announcement phase, group members enter the message posting phase. A client's message posting process is in principle a public key signature verification procedure. A given client signs her message using her private key, and then submits it to her upstream server. The server verifies the signature using a public key from the fresh pseudonym list. If the verification succeeds, the server posts the message and associates the corresponding reputation score with the message. This works, because each client has been assigned a "temporary" public-private pair, $\langle pk_{\pi(i)}, y_i \rangle$ based on g_{m+1} in the announcement phase (§5.3).

AnonRep uses ElGamal signature scheme for message verification. Suppose some client i wants to post a message m. She first chooses a random k so that $1 < k < p-1$ and $\gcd(k, p-1) = 1$. Then, the client computes $r = g_{m+1}{}^{k} \bmod (p-1)$, where g_{m+1} is the final generator obtained from the announcement phase. With r in hand, the client computes $s = (H(m) - y_i \cdot r)k^{-1} \bmod (p-1)$, where y_i is i's private key and $H(m)$ is the message m's hash value. Finally, the client sends her upstream server the signature $\sigma = (r, s)$ and message m through anonymous communication tool (*e.g.*, Tor [19] or Vuvuzela [39]), which can hide the client's local information.

After receiving the message and signature pair (m, σ), the upstream server verifies it by checking $g_{m+1}{}^{H(m)} \overset{?}{=} pk_{\pi(i)}^{r} r^{s}$. If the verification is correct, then the server concludes that the message m was sent by some client whose one-time pseudonym is $pk_{\pi(i)}$. Thus, the server associates m with the reputation score corresponding to $pk_{\pi(i)}$ in the fresh pseudonym list. Such message posting design enables servers to attach the corresponding reputation scores to clients' one-time pseudonyms without learning their long-term identities.

If a client posts multiple messages in the same round, each message would be associated with the same pseudonym. Thus, we suggest that clients post one message in each round to avoid tracking at best-effort.

5.5 Feedback Phase

Finally, group members enter the feedback phase. Clients can provide feedback (either positive or negative) to different messages to indicate the quality of the messages. At the end of this phase, the servers update the reputation of each one-time pseudonym based on the feedback on its messages. Then, the servers perform the announcement phase in reverse to "transform" the updated fresh pseudonym list back to the reputation list consisting of clients' long-term identities and their now updated and again encrypted reputations. The feedback phase could start at the same time as message posting phase or after message posting phase, but should not end before message posting phase.

Feedback collecting. In our design, during a feedback phase, any client can submit her upstream servers her feedback on messages posted by other clients. In various applications, feedback may take a different form. For example, in a message board application like Yelp, feedback consists of votes clients cast while in Twitter, feedback is in a form of following another person. In AnonRep, we use +1 and -1 to denote positive and negative feedback, respectively.

Suppose some client wants to provide some feedback F (*i.e.*, +1 or -1) to a message m, she creates a tuple in the form of $\langle F, m \rangle$, and generates a linkable ring signature for this tuple $\sigma(\langle F, m \rangle)$, by following the signature generation step in §4.2. The client uses anonymous communication tool (the same as in the message posting phase) to send the tuple and the signature to her upstream server. *Note:* When the client generates the linkability tag, she needs to use $t = H(H'(m) + \vec{pk})$ rather than $t = H(\vec{pk})$ mentioned in §4.2, where $H'(\cdot)$ is another public hash function, $H'(m)$ is message m's hash value, and $\vec{pk}$ is the fresh pseudonym list in the current round. The goal of this design is to prevent clients from submitting duplicate feedback on the same message. If a malicious client signs and submits duplicate feedback on the same message m, then this behavior would be detected by the `Check` primitive (in §4.2), because both linkable ring signatures are generated by the same $t = H(H'(m) + \vec{pk})$ and private key. In this case, the duplicate feedback would be ignored by AnonRep. On the other hand, if a client signs feedback on different messages, the generated signatures would be different since they have different hash values.

If the upstream server's verifications succeed (including both verification and linkability checking steps in §4.2), the server associates the received feedback with the message m.

In summary, AnonRep derives two capabilities from linkable ring signatures. First, no member learns who provided feedback on the messages, except that it came from a member of the AnonRep group. Thus, each client's activities remain unlinkable, even if she provides feedback on multiple messages in the same round. Second, if some dishonest client submits duplicate feedback to the same message, such behavior is detected.

Reputation updating. At the end of the feedback phase, the servers update the reputation of every one-time pseudonym based on the feedback received on the messages associated with it. Because the collected feedback is stored in plaintext, AnonRep can utilize diverse reputation algorithms. For example, one-time pseudonym x's message receives 3 positive and 2 negative votes. If x's current reputation score is 4, then AnonRep will update x's reputation to 4 + (3-2) = 5.

After the reputation updates, servers successively perform the *reverse* scheduling protocol to "transfer" the current fresh pseudonym list containing each client's one-time pseudonym and *updated* reputation score back to the reputation list. *Note:* This new reputation list is similar to the input reputation list for the current round, but the encrypted reputation score of each long-term identity in the new reputation list has been updated.

So far, we described one message-and-feedback round, which has updated clients' reputation scores based on their activities while protecting their privacy. The new reputation list would be used as the input for the next round (see Figure 4).

5.6 Practical Considerations

This section presents several practical issues AnonRep faces and possible solutions to address them.

5.6.1 Performance Optimization

The announcement phase and the feedback collecting phase may cause long latencies when the client population is large, *e.g.*, $\geq$ 100,000. This is because the cryptographic primitives used in these two phases – verifiable shuffle and linkable ring signatures – become more computationally expensive as the number of clients grows.

AnonRep addresses this issue by randomly assigning clients into multiple sub-groups, and each sub-group operates in parallel. For instance, with 100,000 clients in total, the original design takes about 15 minutes on each server to run the scheduling protocol. With 20 subgroups of 5,000 clients each, running scheduling protocol on each server takes only 50 seconds thanks to the parallelization. There is a trade-off to make, however. Larger sub-groups provide better anonymity while smaller sub-groups provide better performance.

5.6.2 Misbehavior Detection

Under our trust model, servers may misbehave. There are two possible cases. First, honest servers may notice that some announcement(s) have not been performed correctly by checking the zero-knowledge proofs of correctness generated in the announcement phase when performing the scheduling protocol. It is straightforward for honest servers to check the proofs to detect misbehaving servers. If a proof produced by server j fails, this indicates the server's misbehavior.

Second, clients may find that their reputation scores appear incorrect. There can be multiple causes: a) an upstream server incorrectly attaches the client's reputation score to its posted message; b) the reputation update is performed incorrectly; or c) a reputation is incorrectly initialized during the registration process. In order to detect these types of misbehaviors, the victim client enters into a blame phase where AnonRep randomly selects a witness (*e.g.*, an AnonRep server) to replay the corresponding operations and check all the signatures during the replay. Specifically, for case a) and b), the witness checks the corresponding upstream server who attaches and updates the reputation score, and then identifies whether the server is at fault. Even if the selected witness is dishonest and it does not perform the blame phase properly, no sensitive information is leaked, and the victim client simply needs to re-launch the blame process until she finds an honest witness. For case c), we discuss the solution in §5.6.3.

5.6.3 Registration Verification

The reputation score might be incorrectly initialized during the registration phase, in two ways: 1) a malicious server initializes an incorrect reputation score for a honest newcomer (mentioned in §5.6.2); and 2) a malicious newcomer colludes with a malicious server to assign herself a very high reputation score.

AnonRep can address this problem by asking each server to additionally run a verifiable encryption shuffle [32], which is similar to the version described in §4.1, but only includes the encryption operation. This is because performing a verifiable encryption shuffle enables a server to generate a proof on whether the encryption operation is correctly performed and whether the encrypted value has a desired value (*i.e.*, a correct, initial reputation in this case). In particular, for any newcomer client, each server first adds one ciphertext layer on her initial reputation score, which is a public value in the system setting, then produces a corresponding proof based on verifiable encryption shuffle, and finally sends the above results (ciphertext and proof) to the next server. If some malicious server does not use a correct initial reputation score or does not correctly perform the encryption, then it would be detected by some honest server(s) (at least one).

6 A Security-Enhanced AnonRep

In the design of AnonRep described so far, the reputation scores of AnonRep clients are operated as plaintexts during each round. Such a design, however, may introduce some potential information leakage in certain situations. Suppose in a certain AnonRep group, for instance, a client has a significantly higher reputation score (*e.g.*, 1000) than all the other clients' reputations (*e.g.*, lower than 10). Even though AnonRep enables clients to post messages with different one-time pseudonyms in different rounds, this client's messages could still be tracked across rounds, since her reputation score is too exceptional to hide herself in this group.[1]

The insight on avoiding the privacy leakage through exceptional reputations is to encrypt reputation scores. Thus, we propose a security-enhanced system design called the *reputation budget scheme*. Below we present the design of the security-enhanced AnonRep.

Client registration. When a client i registers, her upstream server S_i generates this client's initial reputation score. Then, all the servers successively encrypt this score using a homomorphic encryption scheme (*e.g.*, ElGamal [20] or Paillier [34]). We use $E_{Hom}(r_i)$ to denote the client i's reputation score r_i which has been homomorphically encrypted by all the servers. Once S_i receives $E_{Hom}(r_i)$, it takes $E_{Hom}(r_i)$ as the input to perform the basic client registration protocol as usual (§5.2), finally obtaining $E(E_{Hom}(r_i))$.

Announcement phase. The servers perform the same announcement phase as in the basic design. Notice however that, the reputations in the generated fresh pseudonym list are no longer in plaintext. Rather, the reputation scores are homomorphically encrypted by all the servers. This means servers in security-enhanced AnonRep no longer can see reputation scores in plaintext.

Message posting phase. When a client i wants to post a message in some round, she may leverage the Camenisch *et al.* proof system [11] or Peng *et al.* [36] to generate a zero-knowledge *reputation budget proof*, PoK, claiming that 1) her actual reputation score is not lower than a budget b, and 2) she wants to use b as the reputation score to post this message. For example, if a client has a reputation score 5, she can use any score no higher than 5 to post her message, *e.g.*, $b = 2$. b is called the *reputation budget*, and is plaintext. *Note:* we just apply the above proof systems [11, 36] as black-box to construct the needed reputation budget proofs. These two proof systems correspond to differnet homomorphic encryption schemes (*i.e.*, ElGamal and Paillier) used during the client registration, respectively.

After generating the reputation budget proof, the client i sends her upstream server a tuple containing the reputation budget and its proof.[2]

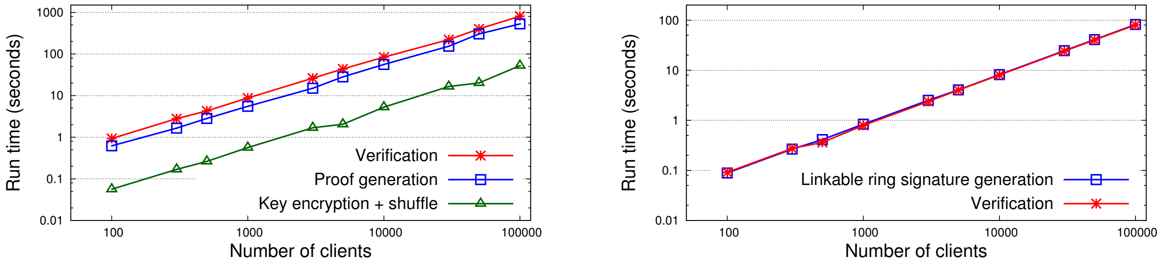
Upon receiving the tuple, the upstream server verifies the proof contained in the tuple. The server learns two things: 1) whether the client i is the owner of her claimed one-time pseudonym; and 2) whether the client i's reputation budget is no more than her actual reputation score. If the verification passes, the server posts the client's message with the reputation budget b.

Because the reputation budget proof is zero-knowledge proof, servers cannot know clients' actual reputation scores. As a result, for a client who has a distinctive reputation score (say, 1000), she can use a relatively low reputation budget (*e.g.*, 5) to post messages, hiding herself in the group. In practice, how to choose an optimal reputation budget depends on the security considerations of specific scenarios.

Feedback phase. Feedback collection is the same as the basic design. Reputation updating is different. In particular, at the end of feedback phase, servers first successively encrypt the received feedback (*e.g.*, votes) leveraging the same homomorphic scheme as used in the registration phase. Servers then update the clients' encrypted reputation scores in the fresh pseudonym list. *Recall:* both the reputation scores in the fresh pseudonym lists and the reputation values from the feedback are encrypted by the same homomorphic encryption scheme. Thus, servers can directly operate the ciphertexts to update the reputation scores for clients due to the homomorphic property.

[1] We make no claim that such situations always happen in reality, because: 1) most reputation systems have an upper bound for reputation scores, and 2) the number of clients with "the highest reputation scores" is normally not too low, as shown in the Stack Overflow dataset [1]. Nevertheless, we still manage to enhance AnonRep to accommodate such situations.

[2] For different zero-knowledge proof constructions, the tuple may be different. For example, for an ElGamal-style construction (*i.e.*, choosing Camenisch *et al.* proof system [11]), the tuple is $\langle R_i, C_i, b, \mathsf{PoK} \rangle$, where C_i is the i's actual reputation in ciphertext (*i.e.*, $E_{Hom}(r_i)$), and b is the reputation budget, R_i is another ciphertext serving for the proof.



(a) Neff verifiable shuffle.

(b) Linkable ring signature.

Figure 6: Microbenchmark (run time) evaluation of cryptographic operations.

This concludes our enhanced system design. Throughout the process, no server learns the actual reputation score of any client, only the reputation budget, thus preserving clients' information even if some clients' reputation scores are significantly different from all others.

7 Implementation and Evaluation

In this section, we first describe our prototype implementation, and then evaluate the prototype.

7.1 Implementation

We have implemented a functional AnonRep prototype. The prototype consists of 2700 lines of Go code as measured by CLOC [18]. Our implementation heavily depends on an open-source Go library of advanced crypto primitives including verifiable Neff shuffle [32], linkable ring signature [26], and various zero-knowledge proofs [11]. More specifically, our prototype implements the complete basic AnonRep design, and all group members in our prototype use UDP to communicate. We support only a limited reputation budget proof construction. The source code of our prototype is available on GitHub.[3]

7.2 Evaluation

Our experiments used NIST QR-512 quadratic residues, although the implementation also works and has been tested with other options such as NIST QR-1024 and QR-2048. We deployed servers in Amazon EC2 as virtual machines. In particular, we used the `c4.8xlarge` instances each with 36 Intel Xeon E5-2666 v3 CPU cores, 60 GB of RAM, and 10 Gbps of network bandwidth. We used laptops as the AnonRep clients each equipped with Intel Core i7 2.6 GHz and 16 GB of RAM.

7.2.1 Microbenchmark

A major performance bottleneck in AnonRep is the overhead of the two cryptographic operations: verifiable shuffle and linkable ring signature.

Figure 6a shows the computational overheads of Neff verifiable shuffle's three main building blocks: 1) key encryption and shuffle, 2) proof generation, and 3) verification. The key encryption and shuffle operation is very efficient, since it only involves simple ElGamal encryptions and element permutations. On the contrary, the proof generation and verification are more expensive, since they need to generate and verify a non-interactive zero knowledge proof, respectively.

Figure 6b shows the run time of generating and verifying linkable ring signatures with different number of clients. Both operations are of very similar cost, and they cost less than 100 seconds even with 10,000 clients. Furthermore, we observe that the computational overheads of both Neff verifiable shuffle and linkable ring signature increase linearly with the number of participating clients.

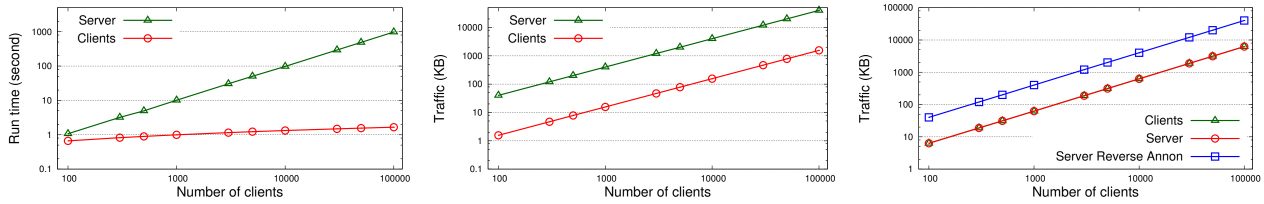
7.2.2 System Overheads

To understand the practicality of AnonRep, we measured server's and client's computational and bandwidth overheads during each phase.

Announcement phase. Figure 7a and Figure 7b show the computational and bandwidth overheads in the announcement phase. Here, each server performs the scheduling protocol, which contains 1) verifying the proof from the former server, 2) encrypting keys and striping one layer from the reputation ciphertext, and 3) generating the proof. To speedup the system, the server performs the proof generation (*i.e.*, step 1 and 2) and the verification (*i.e.*, step 3) in parallel. As shown in Figure 7a, with 100,000 clients, each server needs about 1,000 seconds to execute the scheduling protocol. The computational overhead at client is much less. This is because each client only needs to find its fresh pseudonym whose complexity is $O(\log n)$.

Regarding the bandwidth overhead, each server needs to send its successive server an "intermediate" list containing all clients' keys and encrypted reputation scores, as well as a proof, as shown in Figure 7b. This results in about 40 MB bandwidth overhead if there are 10,000 clients in the network. This is acceptable in practice given the fact that servers are reliably connected. Figure 7b also shows that the client's bandwidth overhead is about 1.5 orders of magnitude smaller than server's

[3] https://github.com/anonyreputation/anonCred



(a) Announcement phase: computational overhead (b) Announcement phase: bandwidth overhead (c) Feedback phase: bandwidth overhead

Figure 7: Comparison of computational and bandwidth overheads between server and client in different phases.

bandwidth overhead. This effectively allows even mobile devices to join our system as the clients.

Message posting phase. In the message posting phase, the only crypto operations are the well-known ElGamal signature generation (client side) and verification (server side). In particular, we want to understand the client's throughput, *i.e.*, how many messages a client can create and sign per second. With different message lengths, we find that a client can generate and sign ten 10MB messages per second, and a server can verify about one hundred 1MB messages per second.

Feedback phase. The feedback phase consists of two steps: feedback collection and reverse scheduling. We mainly measured the overhead of feedback collection, because the overhead of reverse scheduling is the same as the overhead in the announcement phase.

The overheads in the feedback phase are mainly caused by the linkable ring signature operations whose computational overheads have been shown in Figure 6b. Figure 7c shows the bandwidth overhead of each client and server in the feedback phase. We observe that the bandwidth overhead of each feedback is reasonable. For example, in a scenario with 10,000 clients, the bandwidth overheads at the client and server are 500KB and 5MB, respectively.

7.2.3 Practical Deployment

We now discuss and evaluate the AnonRep's deployment in practice, *i.e.*, how to set the duration of each phase and how many servers should be used for an anonymity group. Answering these questions is not straightforward. It depends on specific application scenarios.

In this section, we take Stack Overflow as a sample scenario. We utilize the analytical results from a large Stack Overflow dataset [1]. In particular, we first extract the analytical results conducted by Bhat *et al.* [6], and then discuss how to deploy AnonRep in Stack Overflow based on these results. Finally, we evaluate this AnonRep deployment.

From the measurement study [6], we extract the following features of Stack Overflow: F-1) more than 80% questions receive the accepted answers within 16 hours, and F-2) questions receiving more positive feedback can get accepted answers more quickly (*e.g.*, less than 10 hours).

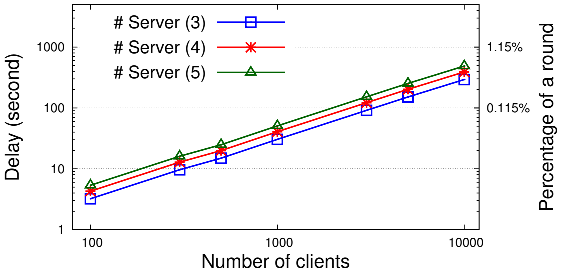


Figure 8: Delay of announcement phase.

According to the feature F-1, we suggest AnonRep set the message posting phase to 16 hours, thus enabling the majority of questions to receive accepted answers within one round. Due to the feature F-2, we allow the feedback phase to start at the same time as the message posting phase in each round. This enables questions to receive answers as soon as possible.

We deployed our prototype on multiple Amazon EC2 `c4.8xlarge` virtual machines, and measured the delay caused by the announcement phase with different numbers of clients and servers. Figure 8 shows that even though we use five servers for the announcement phase, the delay caused by the announcement phase is within 1% of a 24-hour round.

8 Discussion and Limitations

This section discusses some of AnonRep's limitations, and potential solutions.

Intersection attacks via online status. Current AnonRep cannot defend against long-term intersection attacks [24] which target otherwise-honest clients who repeatedly come and go during an interaction period, leaking information to an adversary who can correlate online status with activities across multiple rounds. There is no perfect defense against such intersection attacks when online status changes over time [24]. AnonRep may adopt a *buddy system* [43] whereby a client posts messages and feedback only when *all* of a fixed set of buddies are online. With certain caveats, this discipline ensures that a client's anonymity set includes at least his honest buddies, at the availability cost of making the user unable to transmit (safely) when *any* buddy is offline.

Weighted feedback. Our current design does not support weighted feedback. Namely, all the feedback in our current design has equal impact. Weighted feedback can differentiate clients' feedback. To enable this, the system needs to know the reputation scores of clients who are providing feedback. A possible solution is to introduce another announcement phase between the message posting phase and feedback phase, associating each feedback with its corresponding weight (*i.e.*, reputation score).

Malicious servers. Given the fact that AnonRep applies verifiable shuffle, any malicious servers can be detected and it is possible for an honest server to reveal a malicious server. However, it does not yet have a mechanism to prevent $m-1$ malicious servers from claiming that the other honest server is malicious, since the clients do not know the identity of the truly honest server. Although this can hardly happen as the servers in AnonRep are assumed to be managed by separate operators, it would obviously be better to be able to defeat malicious servers.

9 Related Work

Building an anonymous reputation system is challenging [5, 28]. To our knowledge, AnonRep is the first practical system in this domain.

Electronic cash based schemes. Various anonymous electronic cash (e-cash) protocols [3, 8, 10, 27] have been proposed to maintain the unlinkability of individual users' Peer-to-Peer transactions [13]. Some of them have been applied to build anonymous reputation systems [2, 9, 29]. For example, Androulaki *et al.* proposed RepCoin [2], which attempts to achieve a goal particularly close to AnonRep. However, a general disadvantage of e-cash based anonymous reputation systems, including RepCoin, is that they are incapable of supporting negative feedback, which means reputation of malicious users cannot be confiscated [45]. In addition, e-cash based systems cannot offer fine-grained reputation representations and updates, since all the reputation coins have the same value.

Signature-based approaches. Applying reputation signatures is another approach to preventing users from being tracked. Existing efforts [2, 14, 40] leverage blind signatures to hide the origin and destination of each reputation-based transaction. However, blind signature based approaches heavily depend on the assumption that the centralized authority behaves correctly. Another effort close to our work is to use the signature of reputation [5]. Specifically, each user may express trust in others by voting for them, collect votes to build up her own reputation, and attach a proof of her reputation to any message she posts, while maintaining the unlinkability of her activities. Similar to e-cash based approaches, however, with signature of reputation users cannot express negative feedback, and this approach is also computationally expensive.

Electronic voting based schemes. Electronic voting (e-voting) schemes allow the casting of votes while protecting user privacy [21, 22]. However, e-voting schemes are specifically designed for an election scenario where the candidates have no need to track their historical activities or publish any messages with updated reputation.

Others. Pavolv *et al.* [35] proposed a decentralized system allowing for partial privacy preservation on the user side and easy additive aggregation of users' reputation across the system. A malicious user, however, can easily track other users' activities by assigning specific reputation to victims. In addition, Clauß *et al.* [15] proposed two privacy requirements for reputation systems, *i.e.*, *k*-anonymity and weak rating secrecy. These enable the specification of practical reputation systems for providing strong privacy guarantees.

10 Conclusion

AnonRep is the first practical reputation system which supports regular reputation utilities while maintaining the unlinkability and anonymity of users' historical activities. AnonRep achieves this goal by an elegant integration of cryptographic techniques, *e.g.*, verifiable shuffles and linkable ring signatures, with a multi-provider deployment architecture. The experimental evaluation based on our functional prototype suggests that AnonRep can be applied to existing online services to provide the anonymous reputation utility.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Ratul Mahajan, for their insightful comments. We also thank Mahesh Balakrishnan, Henry Corrigan-Gibbs, Michael J. Fischer, Krishna P. Gummadi, Jon Howell, and Nickolai Zeldovich for their valuable feedback on earlier drafts of this paper. This research was supported in part by the NSF under grants CNS-1149936, CNS-1409599, and CNS-1407454.

References

[1] Stackoverflow dataset. `http://www.ics.uci.edu/~duboisc/stackoverflow/`.

[2] Elli Androulaki, Seung Geol Choi, Steven M. Bellovin, and Tal Malkin. Reputation systems for anonymous networks. In *8th Privacy Enhancing Technologies (PETS)*, July 2008.

[3] Man Ho Au, Willy Susilo, and Yi Mu. Practical anonymous divisible e-cash from bounded accumu-

lators. In *12th Financial Cryptography and Data Security (FC)*, January 2008.

[4] Lars Backstrom, Cynthia Dwork, and Jon M. Kleinberg. Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography. In *16th International Conference on World Wide Web (WWW)*, May 2007.

[5] John Bethencourt, Elaine Shi, and Dawn Song. Signatures of reputation. In *14th Financial Cryptography and Data Security (FC)*, January 2010.

[6] Vasudev Bhat, Adheesh Gokhale, Ravi Jadhav, Jagat Sastry Pudipeddi, and Leman Akoglu. Min(e)d your tags: Analysis of question response time in StackOverflow. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, August 2014.

[7] Johannes Blömer, Jakob Juhnke, and Christina Kolb. Anonymous and publicly linkable reputation systems. In *19th Financial Cryptography and Data Security (FC)*, January 2015.

[8] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In *24th International Conference on the theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2005.

[9] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Balancing accountability and privacy using e-cash (extended abstract). In *5th Security and Cryptography for Networks (SCN)*, September 2006.

[10] Jan Camenisch, Anna Lysyanskaya, and Mira Meyerovich. Endorsed e-cash. In *28th IEEE Symposium on Security and Privacy (S&P)*, May 2007.

[11] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. Technical Report 260, Dept. of Computer Science, ETH Zurich, March 1997.

[12] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981.

[13] Guihai Chen and Zhenhua Li. *Peer-to-Peer network: Structure, application and design*. Beijing: Tsinghua University Press, 2007.

[14] Delphine Christin, Christian Roßkopf, Matthias Hollick, Leonardo A. Martucci, and Salil S. Kanhere. IncogniSense: An anonymity-preserving reputation framework for participatory sensing applications. *Pervasive and Mobile Computing*, 9(3):353–371, 2013.

[15] Sebastian Clauß, Stefan Schiffner, and Florian Kerschbaum. *k*-anonymous reputation. In *8th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, May 2013.

[16] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in Verdict. In *22nd USENIX Security Symposium*, August 2013.

[17] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Conference on Theory and applications of cryptographic techniques (EUROCRYPT)*, May 1997.

[18] Al Danial. Counting Lines of Code. http://cloc.sourceforge.net/.

[19] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the second-generation onion router. In *13th USENIX Security Symposium*, August 2004.

[20] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In George Blakley and David Chaum, editors, *Advances in Cryptology*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1985.

[21] Jens Groth. Evaluating security of voting schemes in the universal composability framework. In *2nd Applied Cryptography and Network Security (ACNS)*, June 2004.

[22] Jens Groth. Non-interactive zero-knowledge arguments for voting. In *3rd Applied Cryptography and Network Security (ACNS)*, June 2005.

[23] Shouling Ji, Weiqing Li, Neil Zhenqiang Gong, Prateek Mittal, and Raheem A. Beyah. On your social network de-anonymizablity: Quantification and large scale evaluation with seed knowledge. In *22nd Network and Distributed System Security Symposium (NDSS)*, April 2015.

[24] Dogan Kedogan, Dakshi Agrawal, and Stefan Penz. Limits of anonymity in open environments. In *Workshop on Information Hiding*, October 2002.

[25] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. In *16th Privacy Enhancing Technologies (PETS)*, July 2016.

[26] Joseph K Liu and Duncan S Wong. Linkable ring signatures: Security models and new schemes. In *Computational Science and Its Applications (ICCSA)*, May 2005.

[27] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from Bitcoin. In *34th IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[28] Tehila Minkus and Keith W. Ross. I know what you're buying: Privacy breaches on eBay. In *14th International Symposium on Privacy Enhancing Technologies (PETS)*, July 2014.

[29] Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Kim Pecina. Privacy preserving payments in credit networks: Enabling trust with privacy in online marketplaces. In *22th Annual Network and Distributed System Security Symposium (NDSS)*, February 2015.

[30] Arvind Narayanan, Hristo S. Paskov, Neil Zhenqiang Gong, John Bethencourt, Emil Stefanov, Eui Chul Richard Shin, and Dawn Song. On the feasibility on Internet-scale author identification. In *IEEE Symposium on Security and Privacy (S&P)*, May 2012.

[31] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *29th IEEE Symposium on Security and Privacy (S&P)*, May 2008.

[32] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *8th ACM Conference on Computer and Communications Security (CCS)*, November 2001.

[33] Lan Nguyen, Reihaneh Safavi-Naini, and Kaoru Kurosawa. Verifiable shuffles: a formal model and a Paillier-based three-round construction with provable security. *Int. J. Inf. Sec.*, 5(4):241–255, 2006.

[34] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology (EUROCRYPT)*, May 1999.

[35] Elan Pavlov, Jeffrey S. Rosenschein, and Zvi Topol. Supporting privacy in decentralized additive reputation systems. In *2nd Trust Management (iTrust)*, March 2004.

[36] Kun Peng, Colin Boyd, Ed Dawson, and Byoungcheon Lee. Ciphertext comparison, a new solution to millionaire problem. In *7th Information and Communications Security (ICICS)*, December 2005.

[37] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *7th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, December 2001.

[38] Stefan Schiffner, Andreas Pashalidis, and Elmar Tischhauser. On the limits of privacy in reputation systems. In *Workshop on Privacy in the Electronic Society (WPES)*, October 2011.

[39] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Scalable private messaging resistant to traffic analysis. In *25th ACM Symposium on Operating Systems Principles (SOSP)*, October 2015.

[40] Xinlei Oscar Wang, Wei Cheng, Prasant Mohapatra, and Tarek F. Abdelzaher. ARTSense: Anonymous reputation and trust in participatory sensing. In *32nd IEEE International Conference on Computer Communications (INFOCOM)*, April 2013.

[41] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Scalable anonymous group communication in the anytrust model. In *European Workshop on System Security (EuroSec)*, April 2012.

[42] David Isaac Wolinsky, Henry Corrigan-Gibbs, Aaron Johnson, and Bryan Ford. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

[43] David Isaac Wolinsky, Ewa Syta, and Bryan Ford. Hang with your buddies to resist intersection attacks. In *20th Conference on Computer and Communications Security (CCS)*, November 2013.

[44] Andrew Chi-Chih Yao. Protocols for secure computations (Extended abstract). In *23rd Annual Symposium on Foundations of Computer Science (FOCS)*, November 1982.

[45] Ennan Zhai, Ruichuan Chen, Zhuhua Cai, Long Zhang, Huiping Sun, Eng Keong Lua, Sihan Qing, Liyong Tang, and Zhong Chen. Sorcery: Could we make P2P content sharing systems robust to deceivers? In *9th IEEE Peer-to-Peer Computing (P2P)*, September 2009.

Mind the Gap: Towards a Backpressure-Based Transport Protocol for the Tor Network

Florian Tschorsch Björn Scheuermann
Humboldt University of Berlin

Abstract

Tor has become the prime example for anonymous communication systems. With increasing popularity, though, Tor is also faced with increasing load. In this paper, we tackle one of the fundamental problems in today's anonymity networks: network congestion. We show that the current Tor design is not able to adjust the load appropriately, and we argue that finding good solutions to this problem is hard for anonymity overlays in general. This is due to the long end-to-end delay in such networks, combined with limitations on the allowable feedback due to anonymity requirements. We introduce a design for a tailored transport protocol. It combines latency-based congestion control per overlay hop with a backpressure-based flow control mechanism for inter-hop signalling. The resulting overlay is able to react locally and thus rapidly to varying network conditions. It allocates available resources more evenly than the current Tor design; this is beneficial in terms of both fairness and anonymity. We show that it yields superior performance and improved fairness—between circuits, and also between the anonymity overlay and concurrent applications.

1 Introduction

Tor [15] is currently the first choice to preserve online privacy. Implementing what has become the standard architecture for low-latency anonymity services, it routes application-layer data, packaged into equally-sized *cells*, along a cryptographically secured virtual *circuit* through an overlay network. Clients build a circuit by selecting three *relays* (an entry, a middle, and an exit) and establishing cryptographic key material with each of them. A circuit can carry data from one or more application-layer streams. In every overlay hop, one "skin" of encryption is added (or removed, depending on the direction of communication). Intermediate relays are neither able to read the cell contents nor to link streams to a specific source and destination at the same time. This is the foundation for achieving anonymity.

Unfortunately, the current overlay design faces major performance issues. Previous work on improving this more often than not focused on isolated symptoms: for instance, cells that dwell for a too long time in socket buffers [22, 37], a too rigid end-to-end sliding window mechanism [6, 46], security threats due to unbounded buffer growth [25], or unfairness effects caused by different traffic patterns [5, 24, 42]. We note that all of the named problems boil down to unsuitably chosen or unfavorably parameterized algorithms for congestion control, scheduling and buffer management. While this has been pointed out before [37, 45], a consistent and superior overall protocol design—beyond treating the symptoms—is still missing.

Designing such a protocol raises interesting challenges, because the requirements in anonymity overlays deviate in several important points from those of congestion control in the general Internet. First, anonymity demands that relays used along one circuit should be located in different legislations and autonomous systems. This implies typically long end-to-end latencies. Consequently, end-to-end feedback loops (typically stretching over three overlay hops) are necessarily slow. At the same time, though, relays in an anonymity network are aware of individual circuits, because they perform per-circuit cryptography. Therefore, stateful processing per circuit at relays is easily possible, also for the purpose of congestion/flow control. This motivates a protocol design that leverages active participation of the relays.

Second, anonymity demands that control feedback must not reveal user identities, neither directly nor indirectly. Therefore, feedback—especially end-to-end-feedback—must be limited and well considered. This is seen as a reason why reliability should not be implemented end-to-end, but instead hop-wise; this matches Tor's current approach.

Third, relay operators donate resources, in particular bandwidth, to the anonymity overlay. The anonymity traffic typically competes with other traffic on the donator's network. To incentivize relay operation, anonymity traffic should therefore not be overly aggressive.

Other desirable properties include the standard set of requirements in networks with multiple independent users, including, in particular, good resource utilization, fairness between users/circuits and reasonable latencies.

Tor currently multiplexes all circuits between a consecutive pair of relays into one joint TCP connection. Circuits carry application-layer data, not transport-layer or network-layer packets; that is, the anonymized TCP connection to the destination server is established by the Tor exit relay, where the payload byte stream is handed over from the circuit to this connection and vice versa. In each relay, cells arriving over one of the inter-relay TCP connections are demultiplexed, kept in per-circuit queues, and then multiplexed again, according to the next outgoing connection for the respective circuits. This design is complemented with an end-to-end sliding window mechanism with a fixed, constant window size. Due to its fixed size, this window, quite obviously, lacks adaptivity. As a result, excessive numbers of cells often pile up in the per-circuit queues and/or in the socket buffers of a relay—that is, in the "gap" between the incoming and outgoing TCP connections. The large number of inter-relay standard TCP connections furthermore results in aggressive aggregate traffic, and thus causes unfairness towards other applications in the same network. Last but not least, multiplexing varying numbers of circuits into one joint TCP connection is also the root of substantial inter-circuit unfairness within Tor [44,45].

In this paper we propose a new design, which we call *BackTap: Backpressure-based Transport Protocol*. With BackTap, we replace Tor's end-to-end sliding window by a hop-by-hop backpressure algorithm between relays. Through per-hop flow control on circuit granularity, we allow the upstream node to control its sending behavior according to the variations of the queue size in the downstream node. Semantically, the employed feedback implies "I forwarded a cell". The circuit queue in the downstream relay is therefore, in essence, perceived as nothing but one of the buffers along the (underlay) network path between the outgoing side of the local relay and the *outgoing* side of the next relay. This includes circuit queues and socket buffers into the per-hop congestion control feedback loop, yielding responsiveness and adaptivity. At the same time, it couples the feedback loops of consecutive hops along a circuit, thereby closing the above-mentioned gap. The result is backpressure that propagates along the circuit towards the source if a bottleneck is encountered, because each local control loop will strive to keep its "own" queue short, while its outflow is governed by the next control loop downstream.

We stick to Tor's paradigm of hop-by-hop reliability, and also to Tor's design decision to tunnel application layer data. However, we implement it in a slightly different way: relays in our architecture have a choice whether to accept or to drop a cell on a per-circuit basis. To this end, congestion control and reliability decisions are shifted to the overlay layer, instead of using TCP between relays. This architecture also avoids reliability-related security flaws as they have been found in Tor [25].

We also do not use a fixed window size, neither end-to-end nor per hop. Instead, we adjust the per-hop window size using an appropriately adapted delay-based congestion control algorithm. In previous applications of delay-based congestion control, first and foremost in TCP Vegas [11], its properties have often been seen as a weakness [1,12]: it is less aggressive than its loss-based counterparts and therefore tends to be disadvantaged in competitive situations. In our approach, this weakness becomes a strength, because the aggressiveness of aggregate Tor traffic can be a significant problem otherwise.

BackTap, including all congestion control and reliability mechanisms, can be implemented on the overlay nodes' application layer, based on UDP transport. Consequently, lower-layer changes are not required. A simulation-based evaluation confirms the benefits of the proposed architecture and demonstrates a huge relief of the network regarding congestion.

Our key contributions are (1) identifying the "feedback gap" as the primary cause of Tor's performance problems, (2) a novel approach to flow control for environments where data is forwarded over multiple overlay hops, (3) a hop-by-hop backpressure design that avoids network congestion with quick, local adjustments and is therefore well suited to long-delay overlay paths, and (4) an in-depth evaluation including a simulation framework for Tor with a specific focus on network aspects.

The remainder of this paper is structured as follows. First, we review related work in Section 2. In Section 3, we discuss the problems and the design space and develop the transport protocol. In Section 4, we evaluate the proposed protocol, before we conclude this paper with a summary in Section 5.

2 Related Work

Since Tor's introduction more than a decade ago [15], it has received significant attention in the research community (and beyond). For obvious reasons, this attention has focused on security and privacy aspects. In recent years, though, performance aspects of Internet anonymity in general and the awareness for network congestion issues in particular have become part of the research agenda.

Performance enhancements have been proposed, for instance, by considering an alternative circuit selection algorithm [3,7,47] or through an adaptive prioritization of circuits [5,24,42]. These research directions are orthogonal to our approach and remain applicable.

The authors of [22, 37] find that cells reside in socket buffers for a long time. In [22], it is suggested to fix this by actively querying all sockets before blindly writing to a socket. Thereby the majority of queued cells is kept in the application layer, so that scheduling on circuit granularity becomes possible with a smaller backlog between data leaving the application and leaving the host. This follows the general intentions of [44, 45]. However, it does not solve the fundamental problem of excessive standing queues due to circuit windows that often far exceed the end-to-end bandwidth-delay product—it only moves these queues to a different place.

Transport-related modifications for Tor have been considered before [6, 8, 17, 27, 33, 37, 46]. A comparative summary of most approaches is provided in [31]. Even though each proposal improves individual aspects, most of them [8, 17, 27, 33, 37] still use the same sliding window mechanism and hence inherit the exact same issues.

As observed by [37], a missing TCP segment carrying data from one circuit will also temporarily stall any other circuit on the same connection until the missing segment has been recovered. This results in head-of-line blocking upon TCP segment losses. The manifest remedy is to use separate (loss-based) TCP connections per circuit [8, 37]. However, as we point out in [45], such a modification would largely increase the (already very high) aggressiveness of the traffic, due to the higher number of parallel loss-based TCP connections. It also does not overcome the fundamental problems with the end-to-end window mechanism and the corresponding feedback gap. In this work, we tackle all these issues.

Only [46] and [6] get rid of Tor's sliding window. The author of [46] builds upon UDP to tunnel an end-to-end TCP connection through the entire circuit. However, while this may be considered a very clean design, it soon comes to its limits because of the long round trip times of a full circuit, which impairs the responsiveness of both reliability and congestion control. Using complex protocols like TCP end-to-end also come at a significant risk of leaking identifying attributes and providing a fingerprint (e. g., via the source port or specific combinations of TCP options), so that end-to-end designs are generally not favorable [8, 45].

The authors of [6] substitute the end-to-end window by a hop-by-hop window, with a scheme adapted from congestion control in ATM networks. However, head-of-line blockings and the choice of window parameters remain open issues. Virtually all transport-related approaches continue to use standard TCP with its built-in congestion control. We instead develop a tailored transport design for anonymity overlays, which eliminates the need for an end-to-end window.

Looking beyond the area of anonymity overlays, the design of a Tor relay resembles a Split TCP setting as it also occurs in performance-enhancing proxies (PEPs): data is forwarded from an incoming to an outgoing TCP connection, linked by an application-layer queue. A survey on PEPs, including case studies, can be found in [10]. Split TCP was originally developed in the context of wireless communication and satellites, but nowadays also finds use in content distribution networks [35]. It basically subdivides an end-to-end TCP connection into a sequence of typically two concatenated connections, where a middlebox (e. g., a wireless access point or a router) acts as PEP.

By terminating the connection at the middlebox and acknowledging data before the actual destination received it, Split TCP, in fact, violates TCP's end-to-end semantics. If desired, this can be avoided by acknowledging data upstream only after it has been acknowledged by the downstream node [48]. In the context of anonymity networks, such a strict adherence to TCP semantics is generally considered unnecessary, though (just like for most practically deployed PEPs). Since Split TCP aims for maximizing the utilization of link capacities, PEPs buffer data and hence congestion might become a problem. As it has been noted before [30], using Split TCP in an overlay network poses particular challenges in this and many other regards. Therefore, even though we focus on the case of anonymity networks, some of our results may also be applied in the area of PEPs and for other overlay designs.

3 The BackTap Design

BackTap performs reliability, in-order delivery and flow control on circuit granularity on the application layer. It can be encapsulated in UDP transport, so that there is no need for modifications to the operating system; of course, a transport-layer implementation of the same concepts in the kernel is, in principle, likewise conceivable, but not pursued here. In fact, the approach to tunnel tailored transport protocols has become more and more widespread in recent years, the likely best-known examples are μTP [41] as used in BitTorrent [41] and QUIC [18] designed for HTTP/2. UDP transport can be combined with DTLS [38] or IPsec to provide message integrity and confidentiality, just like Tor currently uses TLS to secure its TCP-based overlay links.

In this section, we motivate and present the building blocks of our transport approach in detail. In order to emphasize the changes that we propose and to point out the major design challenges in anonymity networks, we use the current Tor design as a reference architecture throughout the discussion.

3.1 Tor's Feedback Gap

Tor implements another instance of data forwarding and transport functionality on the application layer, i. e., on

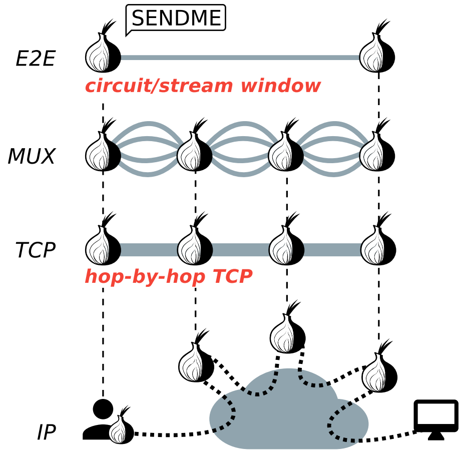


Figure 1: Overview of the layered Tor architecture: relays build a TCP-based overlay, multiplexing circuits, while using an end-to-end sliding window.

top of the existing Internet protocol stack. For this reason, overlay mechanisms will interact with the behavior of underlay protocols and the properties of underlay network paths. This is hardly taken into account in the current design of anonymity overlays in general and of Tor in particular. There are multiple cases where different mechanisms on both layers have overlapping aims. Figure 1 illustrates the various layers of the Tor architecture. The prime example is Tor's end-to-end (E2E) sliding window mechanism between a client's Tor software and an exit relay: it will obviously interact with TCP congestion and flow control, which is used between adjacent overlay nodes. This is also at the heart of the feedback gap in Tor's current design, so that the interplay of these two mechanisms is worth a closer look. This will motivate the key design decisions behind our approach.

Recall, Tor relays forward cells according to the circuit switching principle, but the individual relay does not know about the full path of the circuit. Leaving cryptography aside, relays receive cells over TCP, enqueue them to the respective outgoing circuit queue and then forward them to the downstream node, again via TCP. Between any two adjacent relays, circuits share the same TCP connection. The number of cells in flight for any given circuit is limited by an end-to-end sliding window with a fixed size of 1000 cells (= 512 kB of data).

A node on the receiving side of a Tor circuit signals to send more data by issuing a circuit-level SENDME cell for every 100 delivered cells. Receiving such a SENDME increments the circuit's transmission window by 100. An additional, analogous mechanism exists on the stream level: a *stream* is Tor's notion for the data carried through a circuit, belonging to one anonymized TCP session. Only the end points of a circuit can associate cells with a stream. Intermediate relays, i. e., entry and middle, only differentiate circuits. The stream-level window's fixed size is 500 cells, and stream-level SENDMEs worth 50 cells each are used. Due to the end-to-end sliding window there will be no more than 500 cells in flight on a stream, which is capped by 1000 cells in sum on the circuit level. 1000 cells, though, can be significantly more than the bandwidth-delay product of a circuit, so that long queues build up often: excessive queuing is one of the major causes for huge delays, which Tor painfully experiences [13, 27]. In addition, long queues give implicit preference to bulk flows which constantly keep the queue filled, when compared to more interactive flows, like for instance web traffic.

Even if the end-to-end window size were not fixed (a possible modification which, of course, has been taken into consideration before [6]), the end-to-end delay of a circuit is too high to dynamically adjust it with reasonable responsiveness. Given the specific situation in anonymity overlays, it is fortunately also not necessary to find an end-to-end solution: because intermediate nodes are aware of individual circuits anyway, relay-supported hop-by-hop feedback with local readjustments based on perceived local congestion is a reasonable way out.

What happens, now, if the flow control and congestion control mechanisms of the TCP connections between relays come into play? For the inflight traffic permitted by the end-to-end sliding window, TCP will determine the *local* data flow. Congestion control will adapt to the underlay network path between adjacent relays. Flow control will specifically depend on the receiving relay's policy for reading from sockets.

This is where the feedback gap appears, which we illustrate in Figure 2a: Tor relays read from incoming TCP connections regardless of the current fill level of corresponding circuit queues in the relay. Therefore, limited outflow of a circuit does not propagate back to the incoming side of the relay. For this reason, the end-to-end sliding window with its non-adaptive constant size and its long feedback loop is the *only* mechanism that limits the number of cells in flight along the circuit, and it is the only mechanism that will eventually throttle the source.

One may then, of course, ask whether it would suffice to stop reading from a circuit's incoming socket if a queue for that circuit builds up locally. This, however, is infeasible because, as discussed before, circuits are multiplexed over joint TCP connections. A relay therefore cannot selectively read cells from one specific circuit; stopping to read from one socket could result in massive head-of-line blocking for other circuits.

Using separate standard, loss-based TCP connections per circuit is also not a good design avenue: this would result in excessive numbers of parallel connections, and therefore in very aggressive traffic and high packet loss. In accordance with TCP models such as [34], we argue that more (loss-based) TCP connections imply a smaller rate per connection and thus inevitably a higher packet loss probability *per connection* [45]. In addition, Bufferbloat phenomena [16] cause long reaction times due to excessively large buffers. Thus, approaches such as [8, 37] still suffer from the feedback gap in the same way as Tor does.

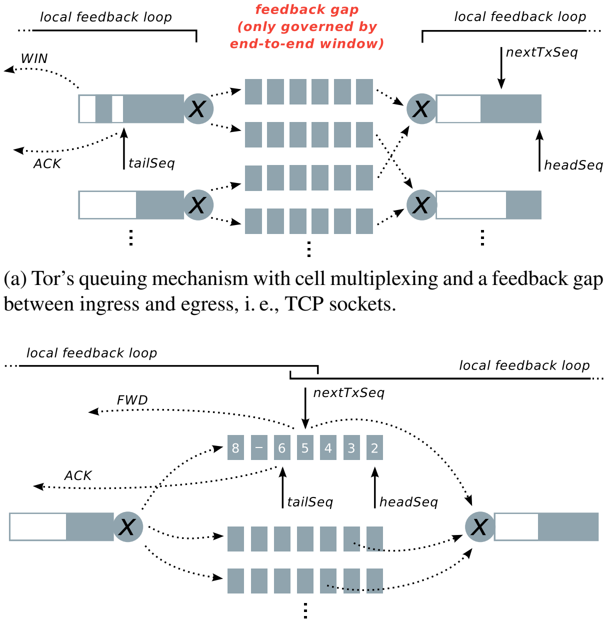
These observations motivate our design based on *delay-based per-circuit congestion control loops*, which can be expected to be much less aggressive than a corresponding loss-based design.

3.2 Realizing Backpressure

A naive realization of the ideas sketched so far—with separate transport-layer connections per circuit, each with delay-based congestion control—would now likely proceed as follows: if an application-layer queue builds up for one circuit, the inflow might be throttled for that circuit by ceasing to read from its incoming connection. The incoming connection's input buffer would consequently fill up, so that the flow control window is not reopened; a zero window would be triggered. This would, in turn, throttle the outflow of the upstream node, so that the outgoing socket buffer fills up. The outgoing socket in the upstream node would then no longer be writable, an application-layer queue would build up there, and so on. Thereby, congestion feedback would propagate indirectly through backpressure.

However, this implies that upstream of the bottleneck, in each relay there must be enough queued data to fill up a) the outgoing socket buffer, b) the application-layer circuit buffer, and c) the incoming socket buffer. Even keeping technical difficulties related to sizing and management of socket buffers in various operating systems aside, incoming and outgoing socket buffers must at least be sufficiently large to cover the bandwidth-delay product of the respective link, in order not to waste performance. Together with the additional application-layer buffer, the total amount of queued data per overlay hop and circuit would once again have to be very significant, and feedback propagation would once again be slow.

To mitigate these effects, we follow a somewhat different, more consequent path: our solution also performs congestion control per circuit, and it likewise does so without multiplexing circuits into joint connections. However, we virtually extend the network into and through the application layer, by emitting flow control feedback only when a cell has been *forwarded* out of the local relay. The application-layer circuit queues in our design therefore take the role of a fused version of the respective ingress and egress socket buffers. Such a queue is illustrated in Figure 2b, and contrasted with the design that is currently followed in Tor, shown in Figure 2a. The feedback gap in the latter is clearly visible, whereas the local feedback loops in our protocol are directly coupled so that backpressure can build up and propagate immediately upon a deterioration of the available bandwidth.



(a) Tor's queuing mechanism with cell multiplexing and a feedback gap between ingress and egress, i. e., TCP sockets.

(b) Fused circuit queue triggers flow control feedback (`FWD`) not until a cell has been forwarded to the successor to achieve backpressure.

Figure 2: Comparison of feedback loops.

In BackTap, arriving cells from the predecessor are read from the UDP socket and processed as usual; that is, in particular the cryptographic operations demanded by the anonymity overlay are performed. The cell is subsequently enqueued in the respective circuit queue. The variable *tailSeq* points to the last cell that has been received in order. *tailSeq* is updated when new cells are received. Cells received out of order may also be queued, with respective gaps in the buffer.

On the other end of the queue, *headSeq* points to the frontmost unacknowledged cell. As soon as we learn that the successor has successfully received the cell, *headSeq* is incremented and the respective cell may be discarded from the buffer.

The third pointer, *nextTxSeq*, is incremented when a cell is forwarded to the downstream relay. The key point that distinguishes our design is: this forwarding at the same time also triggers the transmission of corresponding flow control feedback upstream. In the practical implementation this event triggers the transmission of a `FWD` message. Similar to an `ACK`, an `FWD` carries a sequence number that refers to a cell. The upstream node can make use of `FWD`s to determine a *sending window* (*swnd*) based on the provided feedback. It is allowed to keep at most *swnd* cells in the transmission pipe.

The resulting design is a hybrid between flow control and congestion control: the *swnd* adjustment strategy follows a delay-based approach, based on the latency experienced before receiving an FWD. It will therefore adjust both to the outflow in the downstream node (because only then the FWD is issued) and to the conditions of the network path between consecutive relays (because this path, too, will influence the delays). In essence, it therefore turns the application-layer circuit buffer into yet another buffer along the network path, without a special role from the perspective of the load feedback.

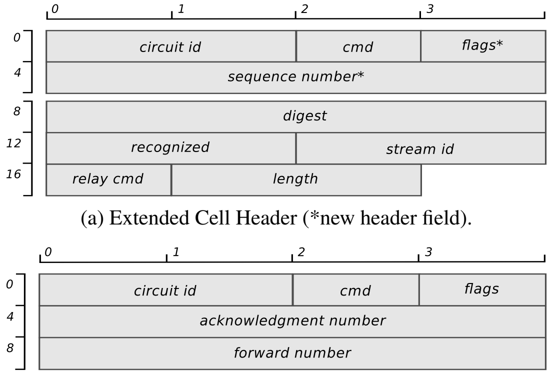
Moreover, tying FWD transmissions to the forwarding of the corresponding cell yields tight feedback coupling between consecutive overlay hops: if the *swnd* adjustment control loop of one overlay hop in a circuit results in a throttled outflow of cells, the FWD arrival delay over the preceding overlay hop will increase accordingly within a one-way local-hop delay. *swnd* can therefore be adjusted quickly. This way, hop-by-hop feedback emerges, and backpressure propagates back to the source. Because delay-based congestion control strives to maintain very short queues, the emerging queues will be small, while available capacity can be fully utilized.

3.3 Reliable Transfer

Tor circuits—or, more precisely, Tor streams—carry application-layer data that expects TCP-like reliable bytestream service. The anonymity overlay design relies on each intermediate hop to ensure reliable in-order delivery. That is, there is no end-to-end ARQ (i. e., reliability/acknowledgment/retransmission) scheme. Reliability on the individual hop in Tor uses the per-hop TCP connections' reliability mechanism; relays are not allowed to drop or re-order cells residing in their per-circuit queues.

We stick to this model also in our proposed transport protocol, i. e., we implement reliability on a per-hop basis. To this end, we use cell sequence numbers to determine the order of cells and to detect losses. The mechanisms generally adhere closely to those employed by TCP. The sender infers, either by a timeout or by duplicate acknowledgments, that cells have likely been lost and retransmits them. The key point where we deviate from TCP's mechanism is where the circuit queue in the downstream node and the coupling between consecutive hop feedback loops comes into play.

The most consequent version of the philosophy of taking the application-layer circuit queue as "yet another network buffer" would use the FWD packets as acknowledgments. This might actually be expected to work reasonably well under many circumstances. However, we argue that it entails a pitfall: after all, when a cell has arrived at the next relay, it is already under the control of the downstream application-layer instance, but reliability feedback is not yet generated. This creates a risk for spurious timeouts, and it might take unnecessarily long to recognize and fix losses.



(a) Extended Cell Header (*new header field).

(b) New Feedback Cell.

Figure 3: Cell structure.

For this reason, as an optimization, we separate reliability on the one hand and congestion/flow control on the other hand in terms of feedback. We provide reliability feedback as early as possible, namely upon arrival of a cell, by sending a corresponding ACK. The calculation of the retransmission timeout (RTO) and the fast retransmit mechanism follow RFCs 6298 [36] and 5681 [4], respectively. Both ACKs and FWDs are cumulative. Handshakes upon circuit establishment and teardown can likewise closely follow their respective counterparts in TCP, and can easily be integrated with Tor's handshakes.

Implementing reliability on the application layer makes it possible to drop arriving cells by a deliberate decision (only before the respective ACK has been sent, of course). This opens up new ways out of a difficult problem: the fact that Tor relays in the current overlay design can be attacked by overloading them with cells which they are not allowed to drop [25]. Dropping excessive cells for a given circuit is a much cleaner and simpler solution than the heuristics that are currently used to relieve Tor from this threatening attack vector.

In an extended cell structure, we introduce new header fields: a sequence number (4 Byte) and a field for flags (1 Byte). They fulfill comparable roles to the respective fields in the TCP header. However, since cells have a fixed size for anonymity reasons, sequence numbers refer to cells rater than bytes. The extended cell header is illustrated in Figure 3a.

For FWDs and ACKs, we introduce a separate message format, much smaller than a Tor cell. Smaller feedback messages can be considered safe and per se do not affect anonymity, because they occur in TCP anyway. Moreover, regular cells and feedback messages—not necessarily for the same circuits—can be encapsulated in one UDP packet traveling between two relays. In a typical MTU up to two regular cells and a number of FWD/ACK messages fit in. The freedom to combine FWDs/ACKs with cells also from other circuits (or, of course, to send them separately if no cells travel in the opposite direction) corresponds to a generalized variant of piggybacking.

Since the Tor protocol already exchanges hop-by-hop control messages and supports variable cell lengths and the negotiation of protocol versions [14], our modifications of the cell structure are resonably easy to integrate. Generally speaking, hop-by-hop feedback messages of any kind and size are allowable under Tor's attacker model, i. e., a local adversary with a partial view of the network. Moreover, our modifications affect the cell preamble only. The preamble is not part of the onion encryption and therefore remains unencrypted on the application layer. Likewise, `FWD`/`ACK` messages are not application-layer encrypted. However, the DTLS encryption between consecutive relays shields the preamble from observers on the wire, and also `FWD`s and `ACK`s. The BackTap design therefore provides *additional* protection in comparison to the current TCP-based transport: when using kernel-level TCP as in Tor today, TCP flow control and ACKs are not encrypted by TLS.

3.4 Window Adjustment

In the proposed protocol design, each node determines the size of its local *swnd* based on the `FWD` feedback from the next hop downstream. `ACK`s are used for reliability, but do not influence the window adjustment.

Most transport protocols, and in particular most TCP variants, use packet loss as an indicator of congestion and therefore as a basis for adjusting their window size or transmission rate; details highly depend on the TCP flavor [1]. Here, we use a delay-based approach as originally used in TCP Vegas [11]. Delay-based congestion control uses latency variations rather than packet losses as a signal for congestion. If queues start to build up—that is, before losses due to exceeded buffers occur—such a control algorithm re-adjusts the congestion window. Thus, they are less aggressive in the sense that they do not force losses and do typically not fully utilize buffers in intermediate nodes.

This reduced aggressiveness constitutes a significant benefit for an anonymity overlay. The Tor overlay at this time is formed by more than 6000 relays (with increasing trend [43]) in a fully connected topology. All currently active connections to other relays compete for the available capacity. The resulting traffic, in sum, is very aggressive and inevitably provokes significant packet loss—also for other traffic traversing the same bottleneck. One may expect that this can significantly be reduced by using delay-based controllers.

Following the ideas of TCP Vegas, we calculate the difference between expected and actual window size as

$$diff = swnd \cdot \frac{actualRtt}{baseRtt} - swnd,$$

where *actualRtt* and *baseRtt* are the RTT with and without load. In the literature they are also referred to as the "experienced RTT" and the "real RTT". We sample the RTT based on the flow control feedback by measuring the time difference between sending a cell and receiving the respective `FWD`. The *actualRtt* is estimated by taking the smallest RTT sample during the last RTT. This reduces the effect of outliers due to jitter on the network path. The *baseRtt* is the minimum over all RTT samples of *all* circuits directed to the *same* relay. Hence, the individual *diff* calculations per circuit use a joint *baseRtt* estimate. This mitigates potential intra-fairness issues of delay-based approaches.

Depending on the value of *diff*, we adjust the sending window every RTT as follows:

$$swnd' = \begin{cases} swnd + 1 & \text{if } diff < \alpha \\ swnd - 1 & \text{if } diff > \beta \\ swnd & \text{otherwise.} \end{cases} \tag{1}$$

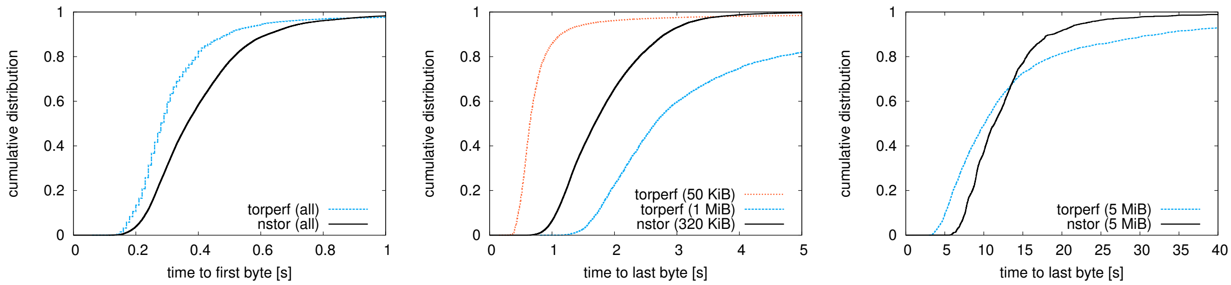
Since *swnd* changes by at most one, it follows an additive increase additive decrease (AIAD) policy. Typically α and β are chosen as 2 and 4 (here measured in cells). Therefore, one may expect that *swnd* does not exceed the bandwidth-delay product by much. This is sufficient to achieve full utilization of the available capacities.

Combined with a locally operating scheduling algorithm that round robins all circuits, this adjustment scheme yields a rate allocation that achieves global max-min fairness between circuits [44], because it aims for maintaining a non-empty queue at the bottleneck. In addition, prioritization heuristics such as [5, 24, 42] can be applied, if a prioritization of certain traffic types and patterns is desired. End-to-end windows and corresponding feedback (in Tor: `SENDME`s) are no longer necessary.

4 Evaluation

A deployment in a real-world anonymity overlay will only be realistic after very thorough preceding evaluations and in-depth discussion in the research community—a process which we hope to initiate with this work. Even deployments in an emulated or testbed-based anonymity network, are also notoriously hard to analyze—because the anonymity itself, of course, prohibits in-depth traceability and measureability. We therefore evaluated the proposed protocol in a large-scale simulation study.

In fact, setting up such a simulation study is a challenging task by itself. As it turned out, there is a missing link in the tool chain when it comes to experimenting with network protocols for Tor under thorough consideration of protocol behavior below the application layer. Some tools focus only on specific aspects, such as the Tor Path Simulator (TorPS) for circuit paths [26]. Others, such as Shadow [23] and ExperimenTor [9], run real



(a) Bulk and web circuits.

(b) Web circuits.

(c) Bulk circuits.

Figure 4: Calibration of the simulation evironment.

Tor code, which results in a high degree of realism regarding the application logic, but at the same time requires extensive development efforts to evaluate experimental protocols [28]. While all approaches have their benefits and drawbacks [40], all miss the "noise" of the real Internet and have no real user behavior. Therefore, assumptions about traffic patterns and user behavior are inevitable anyway. The opportunities, though, to vary parameters, scale the setting, and prototype experimental protocols are much more favorable with advanced network simulators such as ns-3 [19].

Therefore, as a further contribution of this work, we introduce *nstor*, a Tor module for ns-3. It is modeled along the lines of the original Tor software, but clearly focuses on the network aspects. In particular, it includes the transport protocol, the cell transmission scheduler, the traffic shaper (Tor includes a token bucket which can be used by a relay operator to constrain the rate and burst that is used), and the multiplexing. First and foremost, it allows direct and reproducible comparisons of protocol design alternatives in both toy examples and larger scenarios. In addition, with ns-3 the Linux kernel can be hooked, so that practically deployed and widely used transport protocol implementations can be used in the simulations, for additional realism. The code of nstor is publicly available on Github[1].

The key point in setting up an environment for a valid evaluation of Tor is to model the overlay appropriately. The Tor model from [21] serves as a guideline here. In our simulations, we use a star topology for simple, easy-to-analyze toy scenarios, and a dumbbell topology for larger-scale, more realistic experiments. Since approximately 93 % of all Tor relays are currently hosted in North America or Europe [43], the dumbbell topology can be thought to approximate the geographical clustering. For this reason, we adjusted the delay according to the iPlane [29] RTT measurements and the client access rates according to Akamai's state of the Internet report [2] by inverse transform sampling, i. e., generating random samples from its cumulative distribution function (CDF). In addition, we scaled and sampled the Tor consensus (as of 2015/08/04) and generated a large set of circuit paths by feeding this consensus to TorPS [26]. Unless otherwise specified, we assumed neither the physically available bandwidth of the relays' access link nor the Internet backbone to be a bottleneck, but that the relay capacity is bounded by the operators using the above-mentioned token bucket rate limiter.

In accordance to the model proposed in [21], we deliberately distinguish only two types of circuits, bulk and web circuits. Bulk circuits continuously transfer 5 MiB files, i. e., after completing such a download they immediately request another one. Web circuits request 320 KiB files with a random "think time" of 1 to 20 seconds between consecutive requests. Although apparently being very simplistic, it is the common approach used by the Tor community and hence increases the comparability to related research. As [21] stresses, the ratio of simulated web and bulk circuits in relation to the number of relays requires calibration to produce network characteristic that approximate Tor. Therefore, we used the publicly available torperf data set [43], which consists of measurements of various file downloads over the live Tor network. The time-to-last-byte (TTLB) and time-to-first-byte (TTFB) results (as of August 2015) are shown in Figure 4 as CDF plots. For our analysis in a larger setting, we observed that a scenario with 100 relays and 375 circuits with 10 % bulk circuits approximates Tor's performance reasonably well (cf. Figure 4). This configuration corresponds to one of Shadow's example scenarios (as of Shadow v1.9.2). In this setting, we simulated a period of 300 seconds (simulation time), started the clients at random times during the first 30 seconds and left the system another 30 seconds lead time before evaluating. For statistically sound results, all simulations in this paper were repeated with varying random seeds and are presented either with 95 % confidence intervals or as cumulative distribution functions.

In addition to "vanilla" Tor and our approach, BackTap, we also implemented the N23 protocol as proposed

[1] https://github.com/tschorsch/nstor

in [6] and PCTCP [8] (which is conceptually identical to TCP-over-DTLS [37]). This constitutes the first qualitative comparison among alternative transport design proposals for Tor. It also underlines the flexibility of nstor, our ns-3-based simulation module.

4.1 Steady State

First, we take a look at the steady state behavior, i. e., when long-term flows reach equilibrium. For the analysis of the steady state, we focus on the cumulative amount of delivered data of a circuit: by $W(t)$ we denote the amount of payload data delivered to the client up to time t. The counterpart on the sender side is $R(t)$, which denotes the cumulative amount of data injected into a circuit up to time t. Obviously, both functions are non-negative, non-decreasing and $R(t) \geq W(t)$ must hold true at all times.

Given R and W, the end-to-end backlog can be defined as $R(t) - W(t)$, the end-to-end delay as $t_2 - t_1$ for $t_1 \leq t_2$ and $R(t_1) = W(t_2)$, and the achieved end-to-end data delivery rate during an interval $[t_1, t_2]$ as

$$\frac{W(t_2) - W(t_1)}{t_2 - t_1}.$$

Intuitively, these are the vertical difference, the horizontal difference and the slope of the respective functions. For our simulation, we sampled $R(t)$ and $W(t)$ at the sender side (in Tor often called the "packaging edge") and the receiver side (the "delivering edge") of a circuit every 10 ms (simulation time). After the steady state is reached, we performed a linear regression on our data points and calculated the rate, backlog and delay accordingly. The results for a single circuit with a bottleneck rate of 1 500 kB/s (enforced through an application layer limit at the middle relay) and varying end-to-end RTT are given in Figure 5 as a mean of 20 runs with 95 % confidence intervals.

Since Tor has a fixed window size that it will fully utilize, the results with the standard Tor protocol heavily depend on how this window size relates to the bandwidth-delay product (BDP), and thus to the end-to-end RTT. In our example, the circuit window size matches the BDP at an RTT of approximately

$$1\,000 \cdot \frac{512\,\mathrm{B}}{1\,500\,\mathrm{kB/s}} \approx 341\,\mathrm{ms}.$$

Before this point, the backlog significantly increases the delivery delay; for higher RTTs, the download rate drops and asymptotically converges to zero, because the window does not suffice to fully utilize the available bandwidth. This clearly demonstrates Tor's fundamental problem: on the end-to-end level, the only control mechanism is the fixed window, which, however, does not adapt to the network path.

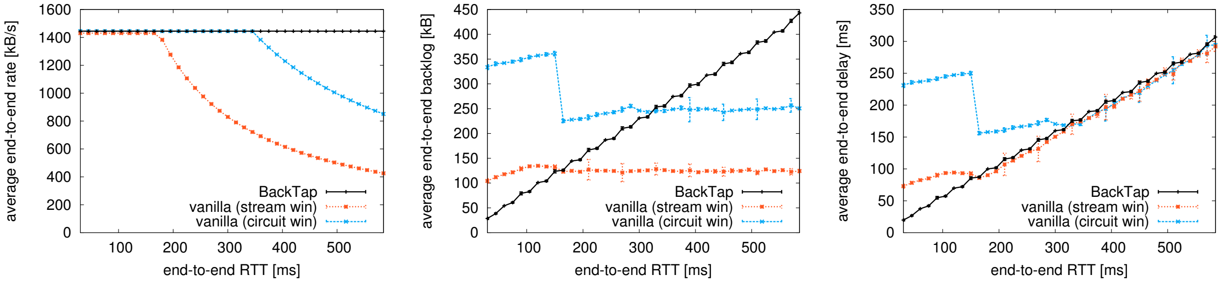
There are some noteworthy phenomena that might be confusing at first sight. In a first approximation according to theory, one would expect that half of a circuit window's worth of data (i. e., approx. 250 kB) is travelling in downstream direction, while the other half of the window is on its way back in the form of SENDME cells. The end-to-end backlog (as defined above: the difference between the amount of sent and received data at a given point in time) should therefore be approximately 250 kB. However, recall that the rate limit is enforced on the application layer by a token bucket. Our model follows the implementation in Tor, where this token bucket is refilled periodically every 100 ms. The bottleneck operates at its capacity limit, always draining its bucket and sending corresponding cell bursts. Thus, about every 100 ms approximately $1\,500\,\mathrm{kB/s} \cdot 100\,\mathrm{ms} = 150\,\mathrm{kB}$ (300 cells) arrive at the client, consequently triggering three SENDMEs. As a result, as long as the RTT is lower than the 100 ms refill interval, only three SENDMEs are on the way back, so that the upstream amount of data is correspondingly higher (approx. 350 kB). For higher RTTs, the observed backlog approaches the theoretical limit without this effect, i. e., 250 kB. Both levels, 350 kB and 250 kB, can be observed in Figure 5b for vanilla Tor ("circuit win").

The respective end-to-end delay, as seen in Figure 5c, behaves according to the built up backlog. That is, while the circuit window is larger than the BDP, there is a noticable delay. Ideally, the end-to-end delay should be half the end-to-end RTT, though.

It is important to note that with a fixed window size there is only one sweet spot, i. e., the BDP. If this point is not met, either the backlog and hence the delay increases or the circuit becomes underutilized. A heterogeneous and volatile network such as Tor is condemned to yield poor performance when employing a static mechanism.

Of course, the same applies to simulations where the (smaller) stream window is the limiting factor: the rate drops much earlier, at $500 \cdot 512\,\mathrm{B}/1\,500\,\mathrm{kB/s} \approx 171\,\mathrm{ms}$. While the end-to-end RTT is less than 100 ms, the three SENDMEs in upstream direction cause a backlog of about 100 kB, this time slightly less than half the window size. Beyond this point, the results meet theory and the backlog levels at half the stream window, that is 125 kB.

We also observed that Nagle's algorithm [32] can interfere with Tor's window mechanism. In a nutshell, Nagle's algorithm suspends transmission for a short period and tries to combine small chunks of data to reduce the overhead. This behavior causes extra delays upon transmission of SENDMEs, and thereby artificially increases the experienced RTT. As a consequence, the rate drops much earlier and the backlog settles at a lower level accordingly, because a larger fraction of the window is spent on the upstream (SENDME) direction (not shown in the figure). However, as soon as scenarios become more

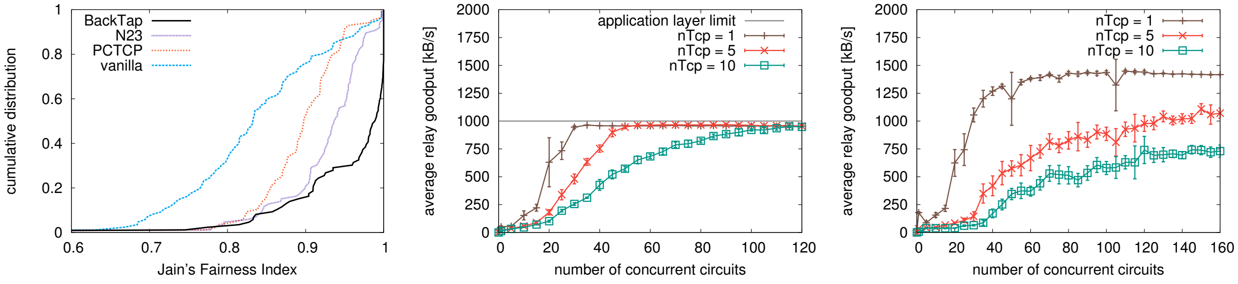


(a) End-to-end rate.

(b) End-to-end backlog.

(c) End-to-end delay.

Figure 5: Single circuit scenario clearly demonstrates Tor's fundamental problem and the benefits of our approach.



(a) Intra-fairness.

(b) Inter-fairness (application layer bottleneck).

(c) Inter-fairness (access link bottleneck).

Figure 6: Fairness evaluation.

complex including more traffic flows, the effect vanishes. By default Nagle is enabled in today's deployments and hence also in Tor. Therefore, we disabled it only to make the previous simulations more easily comprehensible; in all our following simulations Nagle will be enabled. Nevertheless, either with or without Nagle enabled or with the stream or circuit window in place, a fixed-size window is not able to adapt and obviously comes at a severe cost in performance.

In contrast, our approach is able to adjust to the network in all situations. It maintains the rate, while the backlog increases linearly with the RTT (and thus with the BDP). As a result, we achieve an end-to-end delay that always just slightly exceeds the physical RTT. This is the behavior a good transport protocol should exhibit.

4.2 Fairness

For those readers familiar with delay-based congestion control, a number of typical issues will likely come to mind. In particular, they relate to intra-fairness and inter-fairness. We therefore now assess these aspects.

Intra-Fairness Delay-based congestion control depends on accurate RTT measurements. In particular, "late coming" circuits may suffer from an overestimated *baseRTT*. This leads to intra-fairness issues, i. e., to drawbacks in the competition with other delay-based circuits. We mitigate this issue by sharing *baseRTT* information among circuits directed to the same successor. Thus, circuits established later will still base their calculations on sound *baseRTT* measurements. This is a feature of our approach that becomes possible, because the transport logic is implemented in the application layer.

Furthermore, our approach enables cell scheduling on circuit granularity. This avoids fairness issues due to varying numbers of active circuits multiplexed into one transport layer connection, as described in [44]. Figure 6a shows Jain's fairness index [20] calculated over per-circuit goodputs at the respective bottlenecks. This index quantifies fairness as a value between zero and one, where one means perfect fairness. For this simulation, a star topology with 50 relays and 100 circuits generated according to the real-world Tor consensus were used. We started infinite large downloads (i. e., bulk traffic) over each circuit, where the starting times were randomly distributed during the first 30 seconds. We let the simulation settle for another 60 seconds to reach a steady state before evaluating the mean per-circuit end-to-end rates. The results of 20 runs are given as a cumulative distribution plot. Our approach, in fact, achieves a much fairer distribution than all other protocols, which the larger fraction of higher fairness indices confirms.

In these simulations, we also investigated the overhead by comparing the ratio of the achieved goodput (on the application layer) and the actually transmitted bytes (on the MAC layer), i. e., the throughput. The results, as seen in Table 1, show an insignificant difference of approxi-

Table 1: Overhead and backlog comparison

	protocol	$\frac{\text{goodput}}{\text{throughput}}$ ratio	avg. backlog
100 circs	BackTap	0.89	29 kB
	N23	0.86	112 kB
	PCTCP	0.90	181 kB
	vanilla	0.90	184 kB

Table 2: Completed downloads (#dwnlds) and mean rate

		bulk		web	
	protocol	#dwnlds	avg. rate	#dwnlds	avg. rate
375 circs	BackTap	7 503	587 kB/s	52 102	357 kB/s
	N23	4 563	378 kB/s	49 065	215 kB/s
	PCTCP	5 426	424 kB/s	49 513	223 kB/s
	vanilla	5 493	426 kB/s	49 522	228 kB/s
800 circs	BackTap	12 108	439 kB/s	110 142	302 kB/s
	N23	9 067	346 kB/s	104 641	204 kB/s
	PCTCP	10 388	376 kB/s	105 288	207 kB/s
	vanilla	10 491	382 kB/s	105 276	217 kB/s

mately 1 % compared to vanilla Tor. Note that Tor regualry sends at least one SENDME (512 Byte) cell every 250 kB and produces constantly ACKs on the transport layer, while our appraoch sends a comparable amount of ACKs but emits much smaller flow control feedback messages with a higher frequency. As the results suggest the overhead approximately balances out. We also found that our approach largely reduces the number of in-flight cells in the network: the total backlog is about three (for N23) to six times (for vanilla and PCTCP) lower.

Inter-Fairness One of the most prominent caveats of delay-based approaches is that they are "over-friendly" to concurrent loss-based connections. Basically, they reduce the sending rate before loss-based approaches do, because they detect congestion earlier. In some cases this is an intended behavior (cf. LEDBAT [39]), while in the case of TCP Vegas this was generally perceived as an issue [1, 12]. However, if a number of delay-based sessions come together, they are in sum able to compete well [12]. We exploit the properties of delay-based congestion control, because it allows the anonymity overlay to compete more reasonably with other applications (using loss-based TCP) in the relay operators' networks.

We simulated a scenario with a varying number of parallel circuits (on the x axis) and a likewise varying number of competing loss-based TCP connections (*nTcp*). The TCP connections represent downloads that are performed on the same machine as the Tor relay. In a first setting, we limited the anonymity relay bandwidth to 1 MB/s (by the token bucket), while the access link has twice that capacity. In a second setting, we left Tor virtually unlimited (token bucket configured to 10 MB/s) and let the access link become the bottleneck. For small numbers of circuits, the results in Figure 6b and 6c clearly demonstrate in both settings the over-friendly behavior of the delay-based controller, relative to the number of TCP connections. A higher number of active circuits still leaves a good fraction of the total 2 MB/s for the competing non-anonymity connections. For typical relays today one may expect between a few hundred and several thousand concurrently open circuits [8]; of course, not all of them are active all the time.

We believe that the over-friendly inter-fairness of BackTap constitutes an important incentive for relay operators to donate more bandwidth. Typically, relay operators use Tor's token bucket to impose conservative bandwidth limits on their relays. If Tor, however, will appropriately reduce its bandwidth consumption while another application's traffic demand temporarily increases, relay operators will be more willing to operate a Tor relay with less restrictive bandwidth limits. In addition performance penalties of loss-based protocols in environments like Tor [45] will be mitigated.

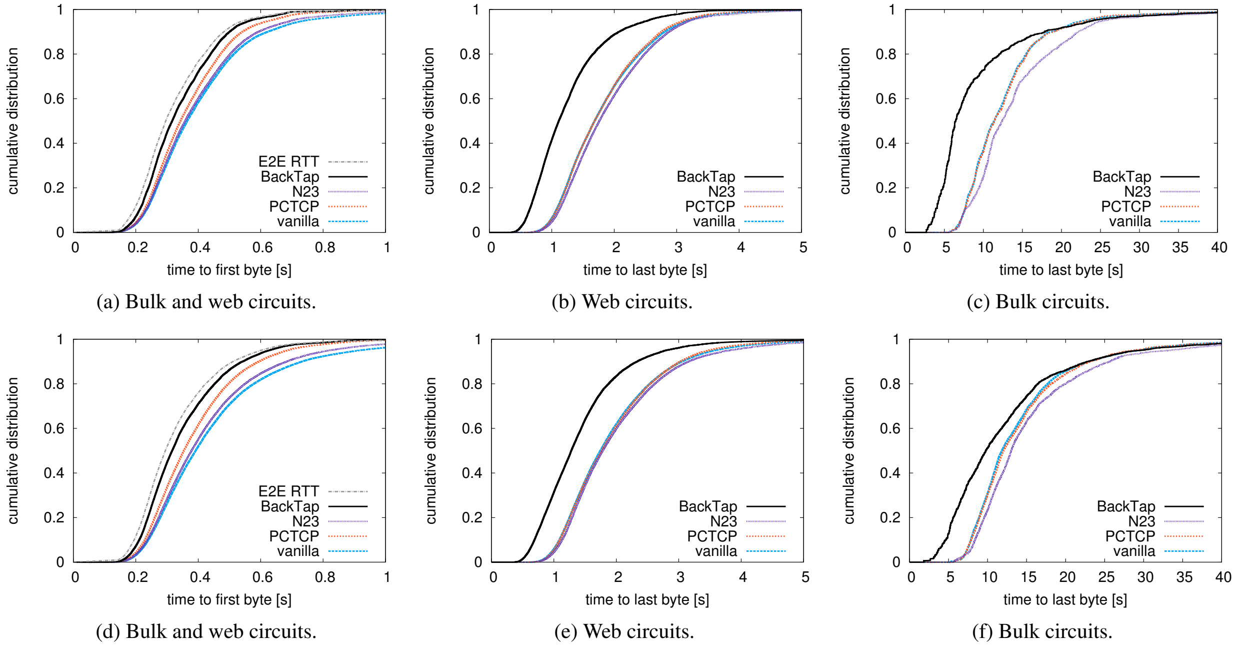
4.3 Larger-Scale Analysis

For an analysis in a larger setting, we simulated scenarios with a dumbbell topology and paths generated according to the real-world Tor consensus, as described above. The time-to-first-byte and time-to-last-byte results of the calibrated setting are shown in Figure 7 (a)–(c) as CDF plots.

In Figure 7a, we show the TTFB results for web and bulk traffic. Virtually all initial byte sequences of answers to requests are delivered faster with BackTap than with any other protocol. In fact, BackTap's TTFB results are very close to the optimum, i. e., the network's physical end-to-end RTT (denoted as "E2E RTT" in the plot). TTFB is an important measure for the interactivity and has a significant impact on the overall user experience. The lower achieved TTFB would likely result in an increased user satisfaction, due to increased reactivity.

The performance gain of our approach becomes apparent when looking at the TTLB results in Figures 7b and 7c. While the download times for web requests typically vary between 1 and 3 s, we achieve significantly better performance, where almost half of all the requests are already completed in less than 1 s. Also the bulk transfers yield better results, i. e. approximately 30 % more bulk downloads are completed in less than 10 s.

In order to assess the performance of our approach in a very congested network, we additionally simulated a scenario with 800 circuits. The results are shown in Figure 7 (d)–(e). Also in this "stress test" scenario, BackTap is able to achieve reasonable results, which in all cases yield shorter download times. Particularly a look at Figure 7f provides a deeper explanation for these results. It shows that the CDF of our approach is closer to the other protocols and "flattens" quicker than in Figure 7c, i. e., more bulk downloads take longer to finish. As a con-



(a) Bulk and web circuits.

(b) Web circuits.

(c) Bulk circuits.

(d) Bulk and web circuits.

(e) Web circuits.

(f) Bulk circuits.

Figure 7: Time to download files over Tor (10 runs, 100 relays, (a)–(c) 375 circuits, (d)–(f) 800 circuits).

sequence, bulk traffic is prevented from "clogging" the network as with the other protocols. However, this does not mean that bulk traffic is treated unfairly: quite in contrast, all of the circuits and flows are treated equally. This is an important feature of our approach: it gives all circuits, web and bulk, a fair share of the network capacity, without the need for (complex, error-prone) explicit traffic pattern analysis and prioritization.

Another perspective on the performance of the various protocols is provided by Table 2. There, we summarize the number of completed downloads (within the simulation time) and the mean download rate for both larger-scale simulation scenarios. In the stress test with 800 circuits, BackTap is able to complete approximately 5 % and 15 % more web and bulk requests, respectively, compared to vanilla Tor. Eventually, the mean download rate is in all cases higher as well. On a more general level, we note that vanilla Tor shows, particularly for the web traffic, a much higher variance of TTFB and TTLB. There is, for instance, always a non-negligible fraction of connections that takes far longer than average. This observation is in line with practical experiences of Tor users and the results presented in [21,43]. Our approach, according to the results presented here, typically reduces the overall variance by more than 17 %.

5 Conclusion

Aware of Tor's fundamental problems and the specific requirements of anonymity overlays, we developed a tailored transport protocol, namely BackTap. In particular, we presented a novel way to couple the local feedback loops for congestion and flow control. It builds upon backpressure between consecutive application-layer relays along a circuit, and a delay-based window size controller. We showed that this can bring a huge relief regarding network congestion by closing the gap between local controllers, so that the need for slow end-to-end control vanishes. In packet level simulations we confirmed the expected improvement.

Besides, there are good reasons why our approach also makes Tor more resilient. First, due to the backpressure, congestion-based attacks will have less influence on other circuits. Second, the much fairer resource allocation makes circuits "look" more "similar", thereby improving the cover traffic properties of concurrent circuits. However, the trade-off between anonymity and performance needs further investigation. In particular, the use of delayed and aggregated feedback to impede traffic confirmation is on our agenda for future work. Generally, we believe that an advanced network traffic control can make Tor's degree of anonymity stronger.

Overall, our approach shows new ways for designing suitable transport mechanisms for anonymity overlays.

Acknowledgements

We thank our students Manuel Rüger and Tobias Schall for valuable discussions. We also like to thank the anonymous reviewers and our shepherd Paul Francis for their constructive, detailed and very helpful feedback.

References

[1] AFANASYEV, A., TILLEY, N., REIHER, P. L., AND KLEINROCK, L. Host-to-host congestion control for TCP. *IEEE Communications Surveys and Tutorials 12*, 3 (2010), 304–342.

[2] AKAMAI. State of the Internet report, Q1 2015.

[3] AKHOONDI, M., YU, C., AND MADHYASTHA, H. V. LASTor: A low-latency AS-aware Tor client. In *SP '12: Proceedings of the 33th IEEE Symposium on Security and Privacy* (May 2012).

[4] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009.

[5] ALSABAH, M., BAUER, K., AND GOLDBERG, I. Enhancing Tor's performance using real-time traffic classification. In *CCS '12: Proceedings of the 19th ACM Conference on Computer and Communications Security* (Oct. 2012), pp. 73–84.

[6] ALSABAH, M., BAUER, K., GOLDBERG, I., GRUNWALD, D., MCCOY, D., SAVAGE, S., AND VOELKER, G. DefenestraTor: Throwing out windows in Tor. In *PETS '11: Proceedings of the 11th Privacy Enhancing Technologies Symposium* (July 2011).

[7] ALSABAH, M., BAUER, K. S., ELAHI, T., AND GOLDBERG, I. The path less travelled: Overcoming tor's bottlenecks with traffic splitting. In *PETS '13: Proceedings of the 13th Workshop on Privacy Enhancing Technologies* (July 2013), pp. 143–163.

[8] ALSABAH, M., AND GOLDBERG, I. PCTCP: per-circuit tcp-over-ipsec transport for anonymous communication overlay networks. In *CCS '13: Proceedings of the 20th ACM Conference on Computer and Communications Security* (Oct. 2013), pp. 349–360.

[9] BAUER, K. S., SHERR, M., AND GRUNWALD, D. Experimentor: A testbed for safe and realistic tor experimentation. In *CSET '11: Proceedings of the 4th Workshop on Cyber Security Experimentation and Test* (Aug. 2011).

[10] BORDER, J., KOJO, M., GRINER, J., MONTENEGRO, G., AND SHELBY, Z. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135 (Informational), June 2001.

[11] BRAKMO, L. S., O'MALLEY, S. W., AND PETERSON, L. L. TCP vegas: New techniques for congestion detection and avoidance. In *SIGCOMM '94: Proceedings of the 1994 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Aug. 1994), pp. 24–35.

[12] BUDZISZ, L., STANOJEVIC, R., SCHLOTE, A., BAKER, F., AND SHORTEN, R. On the fair coexistence of loss- and delay-based TCP. *IEEE/ACM Transactions Networking 19*, 6 (2011), 1811–1824.

[13] DHUNGEL, P., STEINER, M., RIMAC, I., HILT, V., AND ROSS, K. Waiting for anonymity: Understanding delays in the tor overlay. In *P2P '10: Proceedings of the 10th IEEE International Conference on Peer-to-Peer Computing* (Aug. 2010).

[14] DINGLEDINE, R., AND MATHEWSON, N. Tor protocol specification. `https://gitweb.torproject.org/torspec.git/tree/tor-spec.txt`.

[15] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *USENIX Security '04: Proceedings of the 13th USENIX Security Symposium* (Aug. 2004), pp. 303–320.

[16] GETTYS, J., AND NICHOLS, K. Bufferbloat: Dark buffers in the internet. *Queue 9*, 11 (Nov. 2011).

[17] GOPAL, D., AND HENINGER, N. Torchestra: reducing interactive traffic delays over tor. In *WPES '12: Proceedings of the ACM Workshop on Privacy in the Electronic Society* (Oct. 2012), pp. 31–42.

[18] HAMILTON, R., IYENGAR, J., SWETT, I., AND WILK, A. QUIC: A UDP-based secure and reliable transport for HTTP/2. IETF Internet Draft, 2016.

[19] HENDERSON, T. R., LACAGE, M., RILEY, G. F., DOWELL, C., AND KOPENA, J. Network simulations with the ns-3 simulator. In *SIGCOMM '08: Proceedings of the 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Aug. 2008).

[20] JAIN, R., CHIU, D., AND HAWE, W. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. DEC Research Report TR-301, Digital Equipment Corporation, Maynard, MA, USA, Sept. 1984.

[21] JANSEN, R., BAUER, K., HOPPER, N., AND DINGLEDINE, R. Methodically modeling the Tor network. In *CSET '12: Proceedings of the 5th Workshop on Cyber Security Experimentation and Test* (Aug. 2012).

[22] JANSEN, R., GEDDES, J., WACEK, C., SHERR, M., AND SYVERSON, P. F. Never been KIST: tor's congestion management blossoms with kernel-informed socket transport. In *USENIX Security '14: Proceedings of the 23rd USENIX Security Symposium* (Aug. 2014), pp. 127–142.

[23] JANSEN, R., AND HOPPER, N. Shadow: Running Tor in a box for accurate and efficient experimentation. In *NDSS '12: Proceedings of the Network and Distributed System Security Symposium* (Feb. 2012).

[24] JANSEN, R., SYVERSON, P. F., AND HOPPER, N. Throttling tor bandwidth parasites. In *USENIX Security '12: Proceedings of the 21th USENIX Security Symposium* (Aug. 2012), pp. 349–363.

[25] JANSEN, R., TSCHORSCH, F., JOHNSON, A., AND SCHEUERMANN, B. The sniper attack: Anonymously deanonymizing and disabling the tor network. In *NDSS '14: Proceedings of the Network and Distributed System Security Symposium* (Feb. 2014).

[26] JOHNSON, A., WACEK, C., JANSEN, R., SHERR, M., AND SYVERSON, P. F. Users get routed: traffic correlation on tor by realistic adversaries. In *CCS '13: Proceedings of the 20th ACM Conference on Computer and Communications Security* (Oct. 2013), pp. 337–348.

[27] KIRALY, C., BIANCHI, G., AND LO CIGNO, R. Solving performance issues in anonymization overlays with a L3 approach. Tech. Rep. DISI-08-041, Ver. 1.1, Univ. degli Studi di Trento, Sept. 2008.

[28] LOESING, K., MURDOCH, S. J., AND JANSEN, R. Evaluation of a libutp-based Tor datagram implementation. Tech. Rep. 2013-10-001, The Tor Project, Oct. 2013.

[29] MADHYASTHA, H. V., KATZ-BASSETT, E., ANDERSON, T., KRISHNAMURTHY, A., AND VENKATARAMANI, A. iplane: An information plane for distributed services. `http://iplane.cs.washington.edu`.

[30] MAKI, I., HASEGAWA, G., MURATA, M., AND MURASE, T. Performance analysis and improvement of tcp proxy mechanism in tcp overlay networks. In *ICC '05: Proceedings of the IEEE International Conference on Communications* (May 2005), pp. 184–190.

[31] MURDOCH, S. J. Comparison of Tor datagram designs. Tech. rep., Nov. 2011.

[32] NAGLE, J. Congestion Control in IP/TCP Internetworks. RFC 896, Jan. 1984.

[33] NOWLAN, M. F., WOLINSKY, D. I., AND FORD, B. Reducing latency in tor circuits with unordered delivery. In *FOCI '13: Proceedings of the USENIX Workshop on Free and Open Communications on the Internet* (Aug. 2013).

[34] Padhye, J., Firoiu, V., Towsley, D., and Kurose, J. Modeling TCP throughput: A simple model and its empirical validation. In *SIGCOMM '98: Proceedings of the 1998 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Aug. 1998), pp. 303–314.

[35] Pathak, A., Wang, A., Huang, C., Greenberg, A. G., Hu, Y. C., Kern, R., Li, J., and Ross, K. W. Measuring and evaluating TCP splitting for cloud services. In *PAM '10: Proceedings of the 11th International Conference on Passive and Active Measurement* (Apr. 2010), pp. 41–50.

[36] Paxson, V., Allman, M., Chu, J., and Sargent, M. Computing TCP's Retransmission Timer. RFC 6298 (Proposed Standard), June 2011.

[37] Reardon, J., and Goldberg, I. Improving tor using a TCP-over-DTLS tunnel. In *USENIX Security '09: Proceedings of the 18th USENIX Security Symposium* (Aug. 2009).

[38] Rescorla, E., and Modadugu, N. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), Jan. 2012.

[39] Shalunov, S., Hazel, G., Iyengar, J., and Kuehlewind, M. Low Extra Delay Background Transport (LEDBAT). RFC 6817 (Experimental), Dec. 2012.

[40] Shirazi, F., Goehring, M., and Díaz, C. Tor experimentation tools. In *SPW '15: Proceedings of the 36th IEEE Symposium on Security and Privacy Workshops* (May 2015), pp. 206–213.

[41] Strigeus, L., Hazel, G., Shalunov, S., Norberg, A., and Cohen, B. BEP 29: μTorrent transport protocol, June 2009.

[42] Tang, C., and Goldberg, I. An improved algorithm for Tor circuit scheduling. In *CCS '10: Proceedings of the 17th ACM Conference on Computer and Communications Security* (Oct. 2010), pp. 329–339.

[43] The Tor Project. Tor Metrics Portal. `https://metrics.torproject.org`.

[44] Tschorsch, F., and Scheuermann, B. Tor is unfair – and what to do about it. In *LCN '11: Proceedings of the 36th Annual IEEE International Conference on Local Computer Networks* (Oct. 2011), pp. 432–440.

[45] Tschorsch, F., and Scheuermann, B. How (not) to build a transport layer for anonymity overlays. *ACM SIGMETRICS Performance Evaluation Review 40* (Mar. 2013), 101–106.

[46] Viecco, C. UDP-OR: A fair onion transport design. In *HotPETS '08: 1st Workshop on Hot Topics in Privacy Enhancing Technologies* (July 2008).

[47] Wang, T., Bauer, K. S., Forero, C., and Goldberg, I. Congestion-aware path selection for tor. In *FC '12: Proceedings of the 16th International Conference on Financial Cryptography and Data Security* (Mar. 2012), pp. 98–113.

[48] Xie, F., Jiang, N., Ho, Y. H., and Hua, K. A. Semi-split TCP: maintaining end-to-end semantics for split TCP. In *LCN '07: Proceedings of the 32nd Annual IEEE International Conference on Local Computer Networks* (Oct. 2007), pp. 303–314.

Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds

Frank Wang *MIT CSAIL*
James Mickens *Harvard University*
Nickolai Zeldovich *MIT CSAIL*
Vinod Vaikuntanathan *MIT CSAIL*

Abstract

Modern web services rob users of low-level control over cloud storage—a user's single logical data set is scattered across multiple storage silos whose access controls are set by web services, not users. The consequence is that users lack the ultimate authority to determine how their data is shared with other web services.

In this paper, we introduce Sieve, a new platform which selectively (and securely) exposes user data to web services. Sieve has a user-centric storage model: each user uploads encrypted data to a single cloud store, and by default, only the user knows the decryption keys. Given this storage model, Sieve defines an infrastructure to support rich, legacy web applications. Using attribute-based encryption, Sieve allows users to define intuitively understandable access policies that are cryptographically enforceable. Using key homomorphism, Sieve can re-encrypt user data on storage providers in situ, revoking decryption keys from web services without revealing new keys to the storage provider. Using secret sharing and two-factor authentication, Sieve protects cryptographic secrets against the loss of user devices like smartphones and laptops. The result is that users can enjoy rich, legacy web applications, while benefiting from cryptographically strong controls over which data a web service can access.

1 Introduction

A single person often uses multiple web services. Conceptually, the user has one logical data set, and she selectively exposes a portion of that data to each web service. In practice, the services control her data: each service keeps a portion of the user's objects in a walled garden which neither the user nor external services can directly access. Web services often provide user-configurable settings for access control, but the web services (not users) define the semantics of the controls, and users must trust web services to faithfully implement the restrictions. By ceding control of storage to web services, a user also loses the ability to enumerate all of her data, since that data is scattered across a variety of services which hide raw storage via high-level, curated APIs.

Data silos are problematic not only for users, but for *applications* whose value often scales with the amount of user data that is accessible to the application. For example, quantified self applications [62], which track a user's health and personal productivity, work best when given data from a variety of sensors and environmental locations. Similarly, applications which analyze a user's medical records [2] or financial transactions [34] produce the best results when they have access to all of the user's medical or financial data. Unfortunately, the storage silos of modern web services limit a user's ability to share data outside of a silo. For example, wearable fitness-tracking sensors typically upload data to vendor-specific cloud storage, and medical records are often bound to storage that belongs to the medical specialist who measured the data.

The Challenges of Centralized Data: In a user-centric storage model, a user's entire data set would reside in a single, logically centralized cloud store; the user would selectively disclose portions of that data to individual third party applications. Systems like Amber [19] and BStore [20] have explored the benefits of decoupling applications from user data. However, a centralized data store increases the damage that results from a subverted or curious storage provider, because *all* of a user's data is at risk, instead of a service-specific subset.

To protect against untrusted or incompetent storage providers, users can encrypt data before uploading it. However, the ultimate purpose of uploading data is to share it with third party services. Thus, users need a way to selectively expose pointers to encrypted objects (and the associated decryption keys). Protocols exist for sharing cloud data across multiple services, but these protocols have major usability and security problems. For example, the popular OAuth protocol [37] enables cross-site data sharing via delegated API calls (i.e., API calls that act with the authority of a user). However, OAuth policies are invariably defined by web services, not by users. Furthermore, OAuth does not enforce cryptographically strong constraints on the data that delegated APIs can access. So, even if a user could generate her own OAuth policies, she would lack strong assurances about what those policies mean, and how they are enforced.

Given the discussion above, logically centralized storage seems good for users in theory, but difficult to implement in practice. This paper addresses three challenges

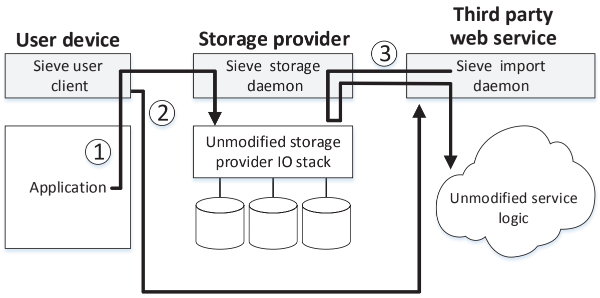


Figure 1: Sieve's high-level architecture. 1) The user uploads ABE-encrypted data to a storage provider. 2) The user generates a data policy for a third party web service. Sieve translates the policy into an ABE decryption key, and sends the key to the web service. 3) The web service pulls encrypted data from the storage provider, decrypts it locally, and injects the data into the unmodified application pipeline.

that emerge from a logically centralized storage architecture. The first is *security*: how do we provide cryptographically strong access controls that protect user data against the compromise of storage providers and user devices? The second challenge is *usability*: how can we express access policies in a way that layperson users will understand, but is translatable to cryptographically enforceable mechanisms? The final challenge is *application richness*: after we have moved user data out of per-application silos and into user-controlled storage, how can we support the complex applications that users currently enjoy?

Our Solution: To address these challenges, we propose Sieve, a new system for delegating access to private cloud data. Figure 1 depicts Sieve's high-level workflow. A user generates raw data on her computational devices, and uploads encrypted versions of that data to a single cloud repository; the user manages and pays for the storage. When a third party web service requests access, e.g., during the first time that a user visits a site, the user generates a high-level access policy (§3.5) for the service. Sieve splits the policy into two pieces: the storage provider learns which objects a third party can access (but not the cleartext versions of those objects), and the third party learns the objects that it can access, and the corresponding decryption keys, while learning nothing about the rest of the user's data set. Once the third party has downloaded the necessary objects and decrypted them, it feeds the cleartext data to a legacy pipeline for handling user content.

Sieve leverages three techniques to implement the workflow in Figure 1:

- Sieve uses **attribute-based encryption** (ABE) [29] to implement cryptographically strong access controls. In ABE, encrypted data is associated with attributes, which are key-value pairs like "date=2012". Decryption keys are associated with policies like the one shown in Figure 2. Policies are defined in terms of attributes and attribute operators like equality and less than. A decryption key can decrypt only ciphertexts whose attributes satisfy the key's policy. Before a user uploads objects to the storage provider, she (or her local device) tags the objects with attributes like the date, the user's current location, or the object type. The uploading device encrypts the objects with the relevant attributes before sending the objects to the storage provider. Later, when a third party web service requests access to the user's data, the user creates a policy for that service. The user's local device translates the policy into an ABE decryption key, and sends the key to the web service. Afterwards, the service uses the key to download and decrypt the subset of user objects that are covered by the key's policy.

```
(type="Fitness" OR type="Medical") AND
  (date > 2012) AND (source="FitBit")
```

Figure 2: Example policy for an exercise application.

- To revoke a third party's access to data, the user informs the storage provider that the third party should no longer be able to download encrypted user objects. However, the third party still possesses a decryption key, and can decrypt leaked ciphertext if the storage server is later compromised. To prevent this scenario, Sieve uses **key homomorphism** [15] to implement revocation. Key homomorphism allows the storage provider to re-encrypt user data without learning the underlying cleartext–the storage provider merely reads the old ciphertext, and overwrites it with the output of a function that accepts the old ciphertext and a user-specified re-keying token as input. Using this *in situ* re-encryption, users avoid the need to re-encrypt data on local devices and then re-upload it. Additionally, if storage providers are honest at the time of key revocation, subsequent storage provider compromises will not reveal data that is encrypted with keys that are revoked (but possibly still in the wild). To the best of our knowledge, Sieve is the first ABE storage system to support re-keying of both metadata and data.
- ABE uses a *master secret key* to generate decryption keys. The loss of this key results in the compromise of the entire cryptosystem. In standard ABE schemes, the master secret is kept by a single trusted authority. In the context of Sieve, this would mean keeping the master secret on a single user device. This is unattractive, since user devices are often lost or stolen. Thus, Sieve uses **secret-sharing** and **two-**

factor authentication to partition the master secret across multiple devices, and prevent unauthorized devices from arbitrarily participating in Sieve's cryptographic protocols.

Sieve represents a middle ground between today's web services (which provide weak user control over data access), and proposed systems from the research community which strengthen user control, but greatly restrict server-side computation [57] or eliminate it altogether [13, 20, 25, 40, 47]. Sieve explores a different point in the design space, one that provides cryptographically strong, user-centric access controls, while still permitting the server-side computation that popular web services require to add value to user data.

To demonstrate Sieve's practicality, we integrated Sieve with two open-source web services. Open mHealth [66] allows users to store and visualize data series for metrics like blood pressure and heart rate; Piwigo [3] is an online photo manager that is similar to Flickr [72]. Integrating Sieve with Open mHealth and Piwigo was straightforward, requiring approximately 200 and 250 lines of code modifications respectively. Experimental results show that the modified systems can handle realistic workloads.

2 Security Goals

We focus on three kinds of principals. A **user** is someone who wants to store data online and selectively expose it to a **third party web service**. The user keeps her (encrypted) data on a **cloud storage provider**. Each user has one storage provider, but potentially many third parties which need delegated access. Potential storage providers include Amazon S3 and Microsoft Azure. Potential third party web services are FitBit, Lark, Mint, and any other application that generates new value from sensitive user data.

The user has a financial agreement with the storage provider: the user pays for the provider to keep her data and participate in the Sieve protocol on her behalf. The user places encrypted data on the storage provider, but never reveals the decryption keys to the provider. This protects the confidentiality of user data if the storage service is malicious or compromised. Using signatures, Sieve also protects the data's integrity. However, Sieve cannot guarantee the availability or freshness of the data that the storage provider delivers to a web service. If desired, Sieve can be layered atop storage systems like CloudProof [55] which do provide those properties.

Sieve does not hide access patterns or object metadata (i.e., ABE attributes) from the storage provider. Thus, a curious provider can learn which encrypted objects a third party has been authorized to read, as well as the attributes that are associated with those objects. If users are concerned about data leakage via access patterns, they can layer Sieve atop an ORAM protocol [44]. To hide attributes from the storage provider, Sieve could use predicate encryption [38, 61]. However, ORAM and predicate encryption incur heavy performance overheads (§6), so Sieve uses lighter-weight cryptography that reduces service latency at the cost of leaking more metadata. We believe that this trade-off is reasonable for many users and companies, given the importance of low latencies in modern web services [69, 71].

With respect to third party web services, Sieve's goal is to reveal user data only as permitted by the user's disclosure policies. After Sieve transmits information to a third party server, Sieve cannot restrict what the third party does with the data. For example, third parties may cache user data locally, even after the user has revoked access to the canonical versions that reside on her storage server. Third parties can also share decrypted user data with other principals via out-of-band, non-Sieve protocols. Preventing these behaviors is beyond the scope of this paper. However, if a web service shares its user-issued ABE key, Sieve can revoke that key, preventing anyone who possesses the key from using it to access user data through the storage provider (§3.6).

Sieve does not prevent client-side attacks like social engineering [9] or cross-site scripting [54]. Sieve also does not protect against a subverted device that the user believes is functioning properly, e.g., a smartphone that is infected with a rootkit. However, Sieve uses secret sharing to protect system-wide secrets like the ABE master key from the loss of a single device (§3.7).

3 Design

As shown in Figure 1, Sieve consists of three components: a user client, a storage provider daemon, and an import daemon that is run by third party web services. The **Sieve client** runs on each user device. The client provides a GUI for defining high-level access policies, and insulates the user from the low-level management of cryptographic keys and data uploading. The **storage provider daemon** communicates with Sieve clients, writing encrypted user data to cloud storage, and using the data's ABE attributes to build an index. The index allows for fast data retrieval by the **import daemons** which belong to web services. An import daemon receives ABE decryption keys from Sieve clients; each key allows the daemon to decrypt a subset of a user's encrypted data.

In Sieve, there are five types of cryptographic keys, all of which are automatically managed by a user's Sieve client. The per-user **ABE master key** helps to generate the individual **ABE decryption keys** which are given to web services. The user's **ABE public key** encrypts metadata blocks with user-provided attributes; note that a storage provider keeps both data and metadata blocks for a user. A web service's ABE decryption keys determine which metadata blocks can be decrypted by the service (§3.4). Metadata blocks point to data blocks, each of which is signed by a **per-user RSA key**, and encrypted by a **symmetric key** that is contained within the associated metadata block. Importantly, all of these cryptographic operations are hidden from the user. A user merely tags data and generates access policies; the Sieve client transparently converts those high-level activities into low-level cryptographic operations.

From the perspective of a third party, a user's Sieve storage is read-only, i.e., only the user can write new objects and update old ones. Third parties use their own storage for data that is derived from a user's Sieve objects.

3.1 Usage Model

In theory, any web service that imports user data is compatible with Sieve. In practice, certain kinds of web services and user data are easier to integrate with Sieve. Sieve works best with

- data streams that are tightly bound to a particular user, and
- web services that can tolerate those data streams being read-only (and perhaps only partially disclosed).

Examples of Sieve-amenable data streams include demographic information like age and location; financial and medical records; sensor data from quantified self applications; longitudinal, cross-site histories of browsing behavior and e-commerce transactions; and multimedia data streams containing photos, videos, and audio. Examples of web services that consume such data streams are social media applications like Instagram [1], exercise trackers like Open mHealth [66], and financial analysis sites like Mint [34] that require access to a user's spending habits.

Applications like Reddit [4] and StackOverflow [6] are less appropriate for Sieve. In these applications, user data has less standalone value to the owner; instead, most of the value derives from embedding the data in a larger, service-specific context like a Reddit discussion. Web-based email is also an awkward fit for the Sieve model, since email services require mutable, per-user state like mailboxes, but Sieve exports read-only storage. An email service could pull read-only outgoing messages from Sieve storage, and implement the mutable mailbox state on the service's own machines. However, such an architecture would be awkward, since users would have no way to selectively expose *incoming* messages to the mail service.

3.2 Overview of ABE

Attribute-based encryption [29] is a public-key encryption scheme in which a cleartext object is associated with attributes that govern how the object is encrypted. Each decryption key has an access control structure (ACS) which enumerates one or more attributes. An ACS can test an attribute for equality (e.g., `location=‘‘Paris’’`) or comparative value (e.g., `age > 35`). An ACS can also chain those simple tests using `AND`s, `OR`s, and `NOT`s. As shown in Figure 3, a key can decrypt an object only if the key's ACS matches the object's attribute values.

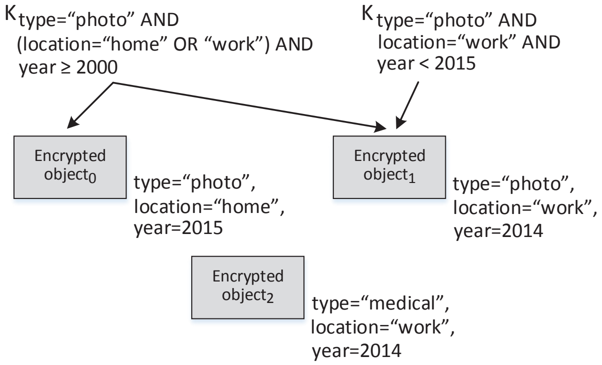

$K_{\text{type="photo" AND (location="home" OR "work") AND year} \geq 2000}$
$K_{\text{type="photo" AND location="work" AND year} < 2015}$
Encrypted object$_0$
type="photo", location="home", year=2015
Encrypted object$_1$
type="photo", location="work", year=2014
Encrypted object$_2$
type="medical", location="work", year=2014

Figure 3: In this example, there are two ABE keys at the top, and three ABE-encrypted objects at the bottom. An arrow indicates that a key can decrypt a particular object.

We use the shorthand notation $K_{a_0,\ldots,a_N}$ to refer to an ABE key whose ACS contains the attribute tests $a_0,\ldots,a_N$. All tests are implicitly joined via `AND`s unless we explicitly note otherwise.

3.3 Assigning Attributes to User Data

Raw user data comes from a variety of sources. Some of it is directly generated by a user's devices; for example, a user might enter financial information directly into a spreadsheet. Data might also come from an external source, like an email attachment. Sieve associates each object, regardless of its provenance, with a set of attributes.

Some attributes can be automatically assigned by hardware, like the GPS coordinates for a running route. Other attributes can be extracted by software, using application-specific transducers or semantic file systems [28, 39, 63]. Users can also manually tag objects. Sieve is agnostic to the manner in which attributes are assigned, although our implementation of the Sieve client provides a GUI which simplifies manual tag assignment. The GUI also allows users to retag an object after it has already been encrypted (§3.9).

Users and web services must agree on data schemas, so that web services can meaningfully aggregate and process information from different users. In particular, web services need to know a standardized set of attributes which are associated with various data types. To define this standardized set, Sieve uses FOAF [17] as the schema model for data about human users, and RDF [21] as the schema model for data about user objects.

Each Sieve user has a standardized FOAF record which stores basic scalar information like her name, location, and birthday. Sieve associates each entry in the FOAF record with an ABE attribute; for example, a user's name is associated with the *userName* attribute. Sieve does not upload the actual FOAF record to the storage provider, since the only purpose of the FOAF record is to standardize the metadata that is associated with each user. Instead, Sieve uploads individual FOAF entries, encrypting each one with the associated ABE attribute (e.g., $\langle\text{“Alice”}\rangle_{K_{field=userName}}$, where “Alice” is the encrypted data and $K_{field=userName}$ means that the data can be decrypted only by web services whose ABE keys have access to the *userName* field attribute).

Sieve associates each data object with a type-specific RDF schema. For example, a W2 tax record has attributes for the user's pre-tax income, her number of dependents, and so on. Similar to FOAF records, RDF records are used to define a standard attribute set for each data type. Sieve uploads individual encrypted RDF entries to the storage provider.

Some data objects like images are not decomposable, i.e., each object is disclosed or not disclosed at the granularity of the entire object. For objects like this, Sieve uploads the entire object, encrypting it using the standardized RDF attributes and any manually added user tags. For example, a photo has standard attributes like height and width, and may also possess user-defined tags like “Vacation” or “Family.”

As applications evolve, RDF and FOAF schemas may change. Sieve is compatible with preexisting techniques for synchronizing schema changes across a distributed system [51, 58, 59].

3.4 Uploading Data to the Storage Provider

Suppose that a user wishes to upload a file F that has attributes $a_0,\ldots,a_N$. Before uploading the file, the Sieve client must encrypt the file such that decryption is possible only with the ABE key whose ACS matches $a_0,\ldots,a_N$. The naïve approach is to directly encrypt F with $K_{a_0,\ldots,a_N}$. However, ABE is a form of public key cryptography, and it is significantly slower than symmetric key cryptography. Thus, Sieve uses a *hybrid encryption* scheme, encrypting the file data with a symmetric key, and encrypting the symmetric key with ABE.

The end-to-end upload protocol is the following. First, the Sieve client generates a symmetric key k, and uses that key to encrypt F. Sieve uploads the encrypted $\langle F\rangle_k$ to the storage provider. The storage provider responds with a GUID for the file. Sieve then uploads a *metadata block* for F. The metadata block is an encrypted pointer containing $\langle GUID,k\rangle_{K_{a_0,\ldots,a_N}}$. Only principals which possess keys that match $a_0,\ldots,a_N$ can decrypt the pointer, fetch the object, and decrypt the object.

In the next section, we describe how web services acquire ABE keys. For now, we merely say that users do not share ABE keys with storage providers. Thus, a storage provider cannot inspect the data that it stores. The provider can try to modify the data or produce fake user data, but Sieve clients sign each object with a user-specific RSA key before encrypting the object; the signatures allow web services to detect tampering.

From the perspective of a third party web service, Sieve storage is read-only. However, a user is free to create new objects, delete old ones, and update objects that reside at preexisting GUIDs. If a user's device has cached the GUID and the symmetric key for a particular object, the user can update that object directly, without having to fetch the associated metadata block and incur ABE overhead to decrypt it.

3.5 Defining and Enforcing Access Policies

In Sieve, all user data is private by default, since users must explicitly share ABE decryption keys that provide access to data. When a third party requests access permissions, e.g., upon the first time that a user visits a web site, the user generates an access policy for the site. Policies are defined in terms of attributes, and the Sieve client provides a GUI which makes it easy for users to explore which attributes her data contains, and which objects would be exposed for a given policy. Policies are simple boolean expressions; for example, a web service used by a physician might receive the policy `(fileType=‘‘medicalRecord’’ AND year>2010 AND doctor=‘‘John’’)`.

After the user defines a policy, her Sieve client assembles the ABE master secret (§3.7) and generates an ABE key with the appropriate ACS. The Sieve client then sends the key and the name of the user's storage provider to the remote web service. The message is protected with TLS [22] to ensure confidentiality and integrity.

Later, when the web service desires to access user data, the service does not need to interact with the user. Instead, the service sends an access request directly to the user's storage provider. The request contains a list of the attributes which belong to the data of interest. The storage provider returns the encrypted metadata blocks for the relevant objects. The web service decrypts the metadata, revealing the GUIDs for the requested objects as well as their symmetric encryption keys. The web service uses the GUIDs to fetch the encrypted objects. After decrypting the objects locally, the service feeds the cleartext data into an application-specific data pipeline.

Once this happens, Sieve is uninvolved in the application workflow. Thus, Sieve is compatible with the current web ecosystem which uses third party computation and storage to add value to user data. However, Sieve provides users with cryptographically strong control over the raw data that each service receives. Sieve's access policies also have three attractive properties:

- The number of policies scales with the number of web services that a user shares data with, not the much larger number of objects that she owns.
- Policy generation is decoupled from object generation. At object creation time, users do not have to speculate a priori about whom a new object might be shared with.
- Policies safeguard objects using cryptography, but users are insulated from the details of key management.

Given all of this, we believe that Sieve strikes a good balance between security, usability, and backwards compatibility with current web services.

3.6 Key Revocation

In Sieve, an individual object is encrypted with a symmetric key k; the object's metadata block (which contains k and the object's GUID) is encrypted with ABE attributes $a_0, \ldots, a_N$. A web service caches its ABE key, and it may also cache symmetric keys and GUIDs, to avoid repeated fetches and decryptions of metadata blocks. Caching makes revocation tricky, since a user that wants to revoke a service's access rights cannot force the service to delete cached keys. An honest storage provider can refuse access requests from deprivileged third parties, but if the storage provider is compromised, it can leak data that is encrypted with ostensibly revoked keys that are still in the wild.

To protect against storage server compromise, Sieve revokes keys by *re-encrypting* user data and metadata with new keys that are not shared with the newly deprivileged third party. If the storage provider is honest at the time of re-keying, then even if it is compromised later, it will never leak data that is encrypted with revoked keys. Leveraging homomorphic encryption [27], the storage provider re-encrypts the data locally, using a re-keying token provided by the user. The storage provider learns nothing about the old encryption key, the new encryption key, or the underlying cleartext; the user avoids the need to download, re-encrypt, and re-upload data from her personal devices.

In the rest of this section, we first describe how data is re-encrypted, and then explain how the associated metadata is re-encrypted.

Re-encrypting data: To enable storage providers to re-encrypt data in situ, Sieve employs a key homomorphic pseudorandom function [15, 50]. We define that function F as

$$F(k,x) = H(x)^k$$

where H is a hash function and k is the secret key associated with each object. F is additively key homomorphic, which means that, for two keys k and k', $F(k,x) \cdot F(k',x) = F(k+k',x)$. All operations described in this section are done modulo p, where p is a large prime.

Using F, we define an encryption scheme whose security is similar to that of AES. Like AES-CTR, our new encryption scheme operates on blocks of data, and uses a random counter to convert a block cipher into a stream cipher. In our new scheme, the jth ciphertext block c_j is equal to

$$c_j = m_j \cdot F(k, N+j)$$

where m_j is the jth cleartext block, and N is a public nonce that is equivalent to the initialization vector in AES-CTR. To decrypt, a third party extracts k from a metadata block and performs the following calculation:

$$m_j = c_j \cdot F(-k, N+j)$$

To revoke the ABE key $K_{a_0,\ldots,a_N}$, a user's Sieve client generates a *re-keying token* for each object that is accessible via $K_{a_0,\ldots,a_N}$. For an object encrypted by k, the re-keying token is $\delta = -k + k'$, where k' represents the new encryption key for the object. The client sends δ to the storage provider; this operation is safe because the provider cannot recover k or k' from δ. The storage provider uses δ to compute a new version of each ciphertext block c_j:

$$\begin{aligned} c_{j,new} &= c_j \cdot F(\delta, N+j) \\ &= m_j \cdot F(k, N+j) \cdot F(-k+k', N+j) \\ &= m_j \cdot F(k', N+j) \end{aligned}$$

In this manner, the storage provider re-encrypts objects without learning the encryption keys or the underlying cleartext.

Re-encrypting metadata: Each user device maintains an integer counter called the epoch counter. The counter is initialized to zero, and represents the number of revocations that the user has performed. When a user device generates a new ABE key, Sieve automatically tags the key with an *epoch* attribute that is set to the current value of the epoch counter. The epoch attribute is a standard ABE attribute; until now, we have elided the *epoch* attribute in key descriptions, but we explicitly represent it in this section.

Suppose that a web service possesses the ABE key $K_{a_0,\ldots,a_N,epoch=i}$, where i is a whole number. To remove the service's access permissions, the user first re-encrypts the affected data using homomorphic encryption. The user then increments the epoch counter to $i+1$. Next, the user generates a new metadata block for each re-encrypted object, inserting the new k. The user encrypts the new metadata block and uploads it to the storage provider; the metadata is encrypted using the updated ABE key $K_{a_0,\ldots,a_N,epoch=i+1}$. Finally, the user sends the new ABE key to any non-revoked web services who possess the old version of the key from epoch i (remember that if the user gives multiple web services access to $a_0,\ldots,a_N$, those services will receive the same ABE key).

Additional web services may require new ABE keys, depending on how the attributes in ABE keys overlap. For example, consider two web services: the first possesses $K_{(a_0 \text{ OR } a_1) \text{ AND } epoch=0}$, and the second has $K_{(a_1 \text{ OR } a_2) \text{ AND } epoch=0}$. Both keys grant access to a metadata block with attributes a_1 AND $epoch = 0$. To revoke the first ABE key, Sieve re-encrypts the metadata block using the attributes a_1 AND $epoch = 1$. As a result, the second, non-revoked web service loses access to the block. Thus, Sieve must give an updated key $K_{(a_1 \text{ OR } a_2) \text{ AND } epoch=1}$ to the second service.

When Sieve re-encrypts a data object, the object's GUID stays the same, but its symmetric key changes. This invalidates any cached symmetric keys that are held by web services. So, when a service receives an updated ABE key, the service discards any cached symmetric keys that were decrypted using the old version of the ABE key. Note that object signatures are unaffected by revocation, because signatures are on cleartext data which is unmodified by the revocation process.

Additional details: A Sieve user will often possess multiple devices; for example, a single user might possess a smartphone, a laptop, and a quantified self device like a FitBit. If a user has multiple devices, then the device which initiates a revocation will broadcast the revoked key and the new epoch counter to the other devices. This ensures that the other devices do not use an old epoch number to encrypt new metadata blocks. Network partitions may delay the rate at which devices learn about a revocation, so when a device receives a revocation notice, the device proactively re-keys any data and metadata that it mistakenly encrypted using the revoked key. Each revocation notice has an issue time, which allows devices to identify which data needs re-keying. Computationally weak devices like FitBits can delegate re-keying work to more powerful devices like laptops.

A revocation message is signed by the public key of the device that issued the message. Devices learn about each other's public keys at Sieve initialization time, and later, when the user adds a new device. By signing revocation messages, Sieve prevents arbitrary devices from injecting fraudulent revocation notices.

To the best of our knowledge, Sieve is the first ABE system which supports full re-keying of *both* data and metadata. Prior ABE systems either cannot revoke keys at all [73], or can revoke access only to metadata [10, 52, 65]; in the latter case, data remains encrypted with revoked symmetric keys, leaving that data vulnerable to storage server compromise or negligence.

3.7 Protecting Against Device Loss

At initialization time, Sieve creates an ABE master secret. Sieve uses the master secret to derive the ABE decryption keys that are given to web services. Thus, the entire cryptosystem is compromised if the master secret is lost.

In a straightforward implementation of ABE, each user device has a copy of the master secret. However, portable devices like smartphones and tablets are often lost [70], meaning that a naïve implementation of ABE exposes the master secret to great risk. Even if users encrypt the master secret with a password-derived key [68], users often pick weak passwords [26], giving the master secret weak protection in practice if a device is lost.

To mitigate the impact of lost devices, Sieve uses Shamir secret sharing [60] to partition the master secret across a user's devices. In a (k,n) sharing scheme, the secret is divided across n devices, and k shares are required to reconstruct the secret. In the context of Sieve, this means that when a user device needs to generate an ABE key, the device must first gather $k-1$ shares from other devices. Only then can the device assemble the master secret, generate the ABE decryption key, send the key to a web service, and then delete the local copy of the assembled master secret.

When the master secret is being assembled, Sieve requires the user to explicitly authorize each participating device to release its local share. By default, Sieve uses a k of 2, so this authorization scheme is similar to two-factor authentication [7]—a user cannot generate an ABE decryption key unless she controls two separate devices (e.g., a laptop and a smartphone). This means that, if an attacker finds a single lost device, that device cannot generate the master secret.

Sieve also employs secret sharing to protect the user's RSA signature key. During uploads to the storage provider, the signature key is used to authenticate the client-side of the TLS session. Thus, the storage provider can reject fraudulent upload attempts from arbitrary devices.

Secret sharing protects the ABE master secret and the user's signing key. However, a lost device possesses a device key that is used to authenticate messages from that device. An attacker with a lost device can try to use the device key to subvert the revocation protocol (§3.6). For example, if a malicious lost device can roll back the epoch to a *smaller* number, uncompromised devices will upload new data that can be decrypted with revoked keys. To prevent attacks like this, Sieve relies on the multi-factor authentication that is built into the secret sharing protocol—revocation requires a device to assemble the master secret, and assembling the master secret requires the user to possess multiple devices.

To add or remove devices from the secret sharing scheme, or to change k, the user must invalidate the old shares. To do so, the user must find k devices to participate in a new secret sharing exchange that uses the updated k and n.

Sieve provides no protections against a subverted device that a user believes is not lost or malfunctioning. For example, if a user wants to upload data from the smartphone that she is currently using, and the smartphone has a rootkit, the phone can arbitrarily delete the user's data, upload garbage, or improperly revoke keys.

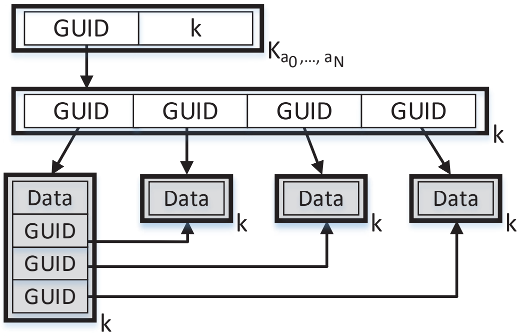


Figure 4: An example of a storage-based data structure. Using indirect GUIDs, the metadata block at the top points to a data block that contains only GUIDs. Those GUIDs point to raw data blocks. Raw data blocks can also embed pointers, as demonstrated by the simple tree structure that links the data blocks.

3.8 Minimizing ABE Overheads

Until now, we have assumed that clients perform two encryptions for every object upload: an ABE encryption for the metadata block, and a symmetric encryption for the data block. ABE is a public key cryptosystem, so ABE operations are much slower than symmetric ones. Fortunately, Sieve clients can use several techniques to reduce the frequency of ABE operations.

The simplest approach is for clients to store multiple objects inside each data block. Creating the associated metadata block will still require an ABE encryption, but subsequent writes and reads of the data block will incur only symmetric cryptography costs—clients can update the data block in-place, without changing the metadata, and third parties can cache the data block's symmetric key to use during reads. For example, a smartphone with a GPS unit might use a single data block to store a month's worth of location data. The phone appends new location samples to the current month's data block, creating a new data block and metadata block when a new month begins.

Clients can also use more complex *storage-based data structures*. For example, as shown in Figure 4, a Sieve client can use indirect GUIDS in the same way that a Unix file system uses indirect data pointers. In this scheme, the top-level GUID for a storage-based data structure refers to a metadata block that points to a *GUID map*. The GUID map is just a data block that contains a symmetric key and additional GUIDs; those GUIDs point to raw data blocks that are encrypted with the symmetric key. Once again, clients eliminate ABE costs by encrypting many objects with the same symmetric key, and caching that key.

Having many, smaller data blocks instead of fewer, larger data blocks is useful if the storage provider does not allow partial block writes (meaning that all writes force the client to upload at least a block's worth of data).[1] Multiple small blocks are also useful if the symmetric cipher does not allow updates to random offsets in the ciphertext.[2]

A raw data block can also embed GUIDs which reference other data blocks. This allows a client to build more complex data structures than flat arrays of data blocks. For example, Figure 4 shows a simple tree with a single parent and three children; by convention, the parent of the tree is the first entry in the GUID map. Each data block can hold multiple items, but when a block fills up, the client creates a new data block, adds the associated GUID to the GUID map, and then updates any internal GUIDs within preexisting data blocks. A third party whose ABE key decrypts the metadata block can traverse the tree structure without additional ABE operations, since all of the data blocks are encrypted with the same symmetric key.

Each storage-based data structure defines a Python API for adding and removing objects, as well as traversing the entire structure. Sieve clients and web services cache the metadata blocks for storage-based data structures, and use the Python APIs to interact with the structures. Thus, Sieve clients and web services are insulated from the low-level details of GUID maps (although both parties can access raw storage if desired, and if Sieve's ABE policies allow such accesses).

Sieve's revocation protocol (§3.6) is compatible with storage-based data structures. When Sieve determines that a metadata block must be re-keyed, Sieve checks whether the metadata refers to a storage-based data structure. If so, Sieve must traverse the structure, identifying

[1] For example, Amazon's S3 allows partial reads, but not partial writes [8].

[2] Many commonly used block cipher modes, such as CBC and CTR, do not easily support new writes to random offsets.

GUIDs and re-keying the associated data blocks. Note that GUID maps are re-encrypted in place, just like any other data block. The revocation protocol does not change the GUIDs that are associated with re-keyed data blocks, so embedded GUIDs inside data blocks remain valid after re-keying.

Each data block that is referenced by a particular GUID map is encrypted with the same k. However, Sieve uses counter-mode encryption [42], and employs a different counter for each block. Thus, if an attacker learns the cleartext for one ciphertext block, the attacker does not have an easier job of decrypting other ciphertext blocks with the same k.

3.9 Relabeling

In Sieve, a user may relabel an object. For example, a user can restrict access by adding an additional attribute to the object. A user can also remove attributes, or swap one attribute for another.

To implement relabeling, a user's Sieve client performs three actions. First, the client replaces the old metadata block on the storage server with a new one that contains a new symmetric key and is ABE-encrypted using the new attributes. Second, the client uses homomorphic encryption to re-encrypt the object under the new symmetric key on the storage server. Finally, the client updates storage-based references to the object, ensuring that the references adhere to the object's new access policy. The client can locate these references because the client knows the old attributes for the object, the new attributes for the object, and the attributes for all of the user's metadata blocks. Thus, the client can determine which references must be patched. For example, suppose that a user has two storage-based data structures S_0 and S_1; further suppose that, due to relabeling, an object must move from S_0 to S_1. By inspecting the object's old attributes, the client determines that the object was originally referenced by S_0. The client homomorphically re-encrypts the object using symmetric key k', traverses S_0 to remove any references to the object, and then adds a $\langle GUID, k' \rangle$ reference for the object to S_1. Sieve performs the traversals, removals, and additions using the APIs defined by the storage-based data structures.

3.10 Discussion

Alternative policy languages: Attribute-based disclosure policies are easy for users to understand, and these policies naturally map to ABE cryptosystems. However, ABE cannot express arbitrarily complex policy functions. Garbled circuits [41] and functional encryption [16] are Turing complete, but they are prohibitively slow. For example, garbled circuits decrypt AES data at a rate that is four orders of magnitude slower than native AES decryption [12]. Relative to functional encryption and garbled circuits, ABE is several orders of magnitude faster.

Paying for storage: In Sieve, each user places her objects in private cloud storage. Someone must pay for that storage. One option is for ad networks to pay. In Sieve, ad networks can be third parties, and they can receive ABE keys to access user data. Using a micropayment system like FileTeller [35], advertisers could pay for the right to collect longitudinal data about a user, and generate targeted advertisements based on that data. By deferring user storage costs, advertisers would encourage users to continue to declassify a subset of their data. Indeed, since each user now stores all of her data in a single place instead of multiple locations, ad networks would gain access to more contextual information than in the current web ecosystem, even if users choose which objects to reveal [67]. Thus, Sieve might enable a happy middle ground in which users gain explicit control over the data seen by third parties, and third parties willingly subsidize private user storage in return for better contextual information.

If ad-driven storage subsidies are poorly designed, they may lead to perverse trade-offs between subsidy amounts and the required levels of data disclosure. A full study of such interactions is beyond the scope of this paper. For now, we merely observe that some users may opt out of the subsidy system entirely. These users will have to pay for their own storage, but there is reason to believe that they would do so. Well-known sites like Pandora, Slashdot, and OkCupid already allow users to pay a small monthly fee to remove advertisements, so there is a pre-existing demographic that is willing to pay money in exchange for better privacy. The popularity of open source applications also demonstrates that developers are willing to make high quality software without the expectation of direct payments from users. Thus, we believe that Sieve's application model is realistic.

Efficient data importing: In the current web ecosystem, users explicitly submit data to web services, making it easy for those services to determine when new information has been created. In Sieve, users submit new data to the storage provider. However, user devices know the tags which are associated with both new data items and web service ABE keys; thus, when a device uploads an object of interest to a particular service, the device can proactively notify the service of the upload.

Storage-based data structures (§3.8) also make it easy for services to identify new data. For example, using a storage-based log, user devices can append new data to the head of the log. A service can cache the GUID and the symmetric key for the log head, and periodically check the beginning of the log for new objects.

Anonymity across services: Some users may not want to be tracked across different web services. For example, a user might be comfortable sharing data with services X and Y, but uncomfortable with X knowing how she interacts with Y, and vice versa. Sieve cannot restrict what services do once they possess user data, so Sieve cannot prevent X and Y from pooling their data and trying to correlate user behavior across both services.

Users can employ various techniques to make tracking more difficult. For example, proxies like Tor [23] allow users to hide their IP addresses from web services. Users can also establish a unique login identity for each web service, or lobby web services to use anonymous credential systems [18]. Unfortunately, Tor and anonymous credential systems rely on network proxies that hurt application responsiveness, and seemingly anonymized data sets can still reveal sensitive user information to machine learning algorithms [24]. Thus, providing anonymity on the web is still an important area for future research.

4 Implementation

Our Sieve prototype consists of a Sieve client, a storage provider daemon, and a Sieve import daemon that is run by third parties. Each component is written in Python, and uses PyCrypto [43] to implement RSA and AES. For ABE operations, we use the libfenc [30] library with elliptic curves [48] from the Stanford Pairing-Based Cryptography library [45]. To build Sieve's key homomorphic symmetric cipher [15], we use the Ed448-Goldilocks elliptic curve library [31].

The storage provider daemon uses BerkeleyDB [53] to store encrypted data blocks, and MongoDB [49] to store metadata blocks. For each data block, the key is a GUID, and the value is a symmetrically encrypted object. For a metadata block, the key is a set of cleartext ABE attributes, and the value is an ABE-encrypted GUID and symmetric key. Metadata blocks are indexed by their attribute fields, and all metadata blocks for a particular user are stored in a MongoDB collection.

The JavaScript code in a web site interacts with the local Sieve client using a small RPC library that we provide. When a web site initially requests access to a user's data, the site's JavaScript sends an `XMLHttpRequest` to a localhost webserver run by the Sieve client. The Sieve client then displays a GUI that allows the user to define an access policy for the site, and send the associated ABE key to the site's web server.

5 Evaluation

In this section, we explore one high-level question: is Sieve practical? To answer this question, we integrated Sieve with two applications. The first was Open mHealth [66], an open-source web service that allows users to analyze their health data. We also integrated Sieve with Piwigo [3], an open-source online photo manager. We show that the integrations were straightforward, and that the end-to-end application pipelines can handle realistic workloads.

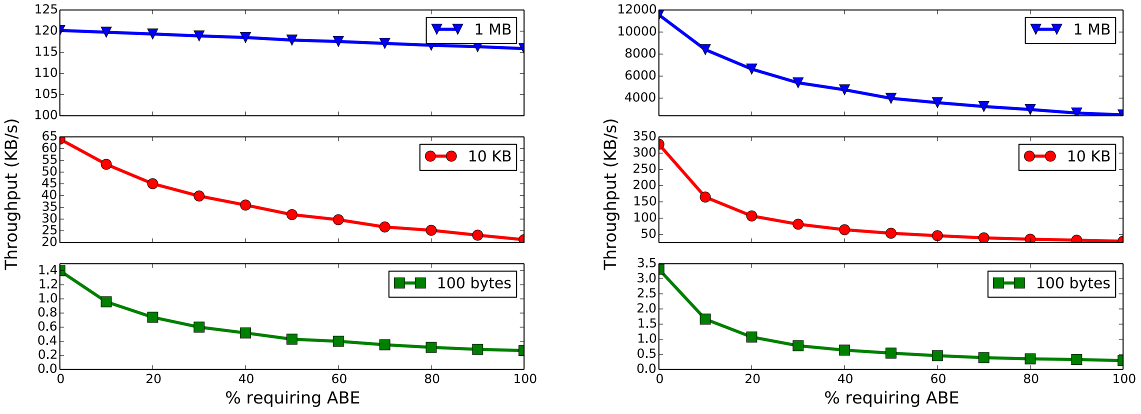
All experiments ran on a 10-core machine with 2.4 GHz Intel Xeon E7-8870 CPUs and 256 GB of RAM. We ran each experiment 50 times, and we report the average (standard deviations were small). Sieve used 2048 bit RSA with SHA256 to sign user objects. ABE operations used 224-bit MNT curves [48]. To symmetrically encrypt objects, Sieve used 128-bit AES in CTR mode, or Ed448-Goldilocks elliptic curves in randomized counter mode. The latter cipher is key homomorphic, but the former is not; by comparing Sieve's performance with these ciphers, we could measure the cost of supporting key revocation (§3.6). All web servers ran on the test machine's loopback interface, to minimize network latency and focus on Sieve's cryptographic overheads.

All GUIDs were 64 bits long. Thus, a metadata block which contained a GUID and an AES key was 24 bytes in size, whereas a metadata block which contained a GUID and an Ed448 key was 64 bytes long.

5.1 Case Studies

Open mHealth: Open mHealth allows users to upload medical data to a web server that will analyze the data and provide explanatory visualizations. To integrate Sieve with Open mHealth, we first modified the Open mHealth client to upload data via the Sieve client instead of directly to the Open mHealth server. We then ran a Sieve import daemon on the Open mHealth web server, configuring the daemon with the data schema used by the Open mHealth analytics engine. These modifications required approximately 200 lines of code to be changed in the Open mHealth platform.

To test the end-to-end performance of the application pipeline, we used Open mHealth's data generator to create a week's worth of health data. The data included information like blood pressure, weight, physical activity, and heart rate. Each day had approximately 14 data points. For each data point, the Sieve client added attributes like the date that the sample was collected, the name of the associated user, and the type of data represented by the sample. The Sieve client used a single storage-based data structure to store the samples for an entire week.



(a) Encryption speed: ABE and Ed448 in randomized counter mode.

(b) Encryption speed: ABE and AES in CTR mode.

Figure 5: Encryption throughput for Sieve, as a function of 1) the size of the data to symmetrically encrypt, 2) the percentage of symmetric data encryptions which also require the ABE encryption of a metadata block, and 3) whether the cipher is AES or key homomorphic Ed448. All experiments assume that each metadata block has five attributes, and each ABE key has 10 attributes. Performance trends for decryption are similar.

The cost for the user to upload the first data point at the beginning of a week was 0.56 seconds; the cost was dominated by ABE encryption. Uploading subsequent data points proceeded at the throughput of the symmetric cipher, requiring 17.1 ms per data point for AES, and 38.5 ms for Ed448.

The mHealth server used the Sieve import daemon to download user data. If the server had no cached GUIDs or symmetric keys, then importing a week of data required 0.49 seconds with AES and 0.78 seconds with Ed448. In this scenario, the server had to download the metadata block, decrypt it with ABE, download the data block, and then decrypt that block using a symmetric cipher. If the server possessed cached GUIDs and symmetric keys, then importing a week of data took only 135 ms with AES, and 469 ms with Ed448.

Piwigo: The standard Piwigo client allows users to upload photos from local storage to the Piwigo web service. We modified the client to upload data to a Sieve storage provider, and we modified the server-side Piwigo code to fetch user data via the Sieve import daemon. These modifications required approximately 250 lines of new Piwigo code.

To test the end-to-end performance, we uploaded a 375 KB photo which had three tags (location, date, and username). If the Piwigo client used AES, the upload required 0.57 seconds if a new, ABE-encrypted metadata block had to be generated. If the client used a storage-based list to avoid the creation of a new metadata block, the upload cost was only 0.06 seconds.

As we explain in more detail in Section 5.2, current Ed448 implementations are slower and less optimized than equivalent AES implementations. Thus, when applications use Ed448, the upload time for a large object is dominated by Ed448 encryption costs, regardless of whether ABE costs are incurred. If the Piwigo client used Ed448, the upload cost for a 375 KB photo was 6.1 seconds if the client also had to generate a new metadata block. By using storage-based data structures to avoid ABE operations, the upload cost dropped to 4.2 seconds. Note that, from the user's perspective, *uploads are asynchronous*. Thus, multi-second upload times are not in the critical path of user-facing activities.

Download times for the Piwigo server demonstrated similar trends. With cached GUIDs and symmetric keys, downloading a photo required 0.14 seconds using AES, and 5.9 seconds using Ed448. Without cached metadata, a download required 0.44 seconds with AES, and 6.3 seconds with Ed448.

Server-side per-core throughput: The storage daemon uses BerkeleyDB to store data objects. The daemon logic is simple, meaning that the daemon can import data at the raw speed of the BerkeleyDB write path. For Open mHealth, the write speed was roughly 50 MB/s per server core, which represented 16,500 users uploading a week's worth of data every second. For Piwigo, the write speed was roughly 200 MB/s per core, corresponding to 550 photo uploads per second (assuming a photo size of 375 KB). Write throughput was better for Piwigo due to BerkeleyDB handling large writes faster than small ones.

Operation	Time
Generating 10 attribute key	0.46 sec
Generating 20 attribute key	0.64 sec
Re-encrypting a metadata block (10 attrs)	0.63 sec
Re-encrypting a metadata block (20 attrs)	0.91 sec
Re-key 100 KB data block	0.66 sec

Figure 6: Computational overheads for key generation and revocation.

We also tested per-core throughput for the import daemon. For Open mHealth using AES, a single core could download and decrypt a week's worth of data for 420 users in one minute; with Ed448, a core could import 70 users' data in one minute. Given a photo size of 375 KB, Piwigo was able to import 235 AES-encrypted photos or 14 Ed448-encrypted photos in one minute. In all experiments, 20% of object imports required the download and ABE-decryption of a metadata block. We believe that 20% is high, since an arbitrary number of objects can be referenced by a single metadata block.

5.2 Microbenchmarks

Encryption speed: Sieve requires clients to symmetrically encrypt each data object before uploading it. Some fraction of uploads will also require clients to ABE-encrypt a metadata block. Figure 5 quantifies the performance of ABE and the symmetric ciphers. For 10 KB objects, pure ABE encryption throughput is 1.1 KB/s, whereas pure Ed448 throughput is 23.8 KB/s and pure AES throughput is 43.5 KB/s. Although clients can perform data uploads asynchronously, in the background, the computational costs for ABE are still quite high. Thus, hybrid encryption (§3.4) and the optimizations from Section 3.8 are crucial for minimizing the number of ABE operations.

For 1 MB objects, the performance gap between AES and Ed448 grows–AES throughput is 12 MB/s, but Ed448 throughput is only 120 KB/s. However, Ed448 is a new elliptic curve, with immature implementations relative to AES. We expect Ed448's performance to improve as its implementations receive more optimization effort.

Key generation and revocation: Figure 6 describes the costs that Sieve pays for generating new ABE keys, re-encrypting metadata blocks, and re-keying a 100 KB data block. The creation of new ABE keys is rare, and occurs only when a new service requests access permissions, or an old service receives modified permissions (possibly as the result of an epoch number increasing after a revocation (§3.6)). During revocation, the metadata blocks associated with the revoked ABE key must be re-encrypted; however, those metadata blocks will typically point to a much larger number of raw data blocks (§3.8), so the overall re-encryption cost of revocation is governed by the speed with which raw data can be re-keyed.

Attribute matching: When the storage provider receives an access request from a third party, the storage provider must locate the metadata blocks whose attributes match those of the access request. Sieve makes the matching process fast by storing metadata blocks in a database that indexes those blocks by their attributes.

Due to space constraints, we omit a full description of matching performance. However, the results are unsurprising, since modern databases are good at building indices. For example, in one experiment, we injected a million metadata blocks into MongoDB; each metadata block had 10 randomly selected attributes from a universe of 35 possible attributes. Then, we submitted access queries in which each query contained 5 random attributes joined with a random set of ANDs and ORs. Each query took 0.13 ms to complete on average.

Secret-sharing: Sieve partitions the ABE master key and the RSA signing key across multiple devices, ensuring that a lost or stolen device will not store a full copy of sensitive cryptographic information. The secret sharing protocol is cheap: ignoring network latency, and assuming that $k = 2$ and $n = 5$, splitting a 2048 bit object like an RSA key requires 0.04 ms, and reconstructing that key requires 0.09 ms.

6 Related Work

Untrusted servers: Browser extensions like ShadowCrypt [32] transparently encrypt the data that a browser sends to unmodified cloud servers. Intentionally encrypted cloud stores like SUNDR [40], Depot [47], and SPORC [25] provide stronger consistency semantics in the face of server-side misbehavior; application logic runs solely on the client-side, over cleartext data, with clients exchanging encrypted data with servers. Other systems that store encrypted data on servers and run application logic on the client-side include BStore [20] and DepSky [13]. All of these systems prevent data leakage due to server compromise or malice. However, these systems are incompatible with applications that leverage server-side computation to add value to raw user data. In contrast, Sieve is totally compatible with server-side computation.

In CryptDB [56], a web application consists of clients, an application server, and a back-end database. The database contains only encrypted data. Using SQL-aware encryption, the application server can execute queries over the encrypted data without revealing cleartext to the database. However, the application server does see cleartext, and can leak user data if compromised. Mylar [57] eliminates the need for an application server, but restricts the encrypted server-side computation to keyword searches. In both CryptDB and Mylar, applications control how user data is shared. In Sieve, user data is decoupled from applications, with users selectively disclosing individual objects to third parties.

Privly [5] allows users to upload encrypted data to a storage server, and share hyperlinks to that data. The hyperlinks can be embedded in sites like a Facebook page, but the hyperlinks reveal no cleartext to the owner of the embedding site. Users register their decryption keys with Privly's browser extension. Later, when the user visits a page and her extension finds a Privly hyperlink, the extension transparently fetches the encrypted data, decrypts it, and rewrites the page's HTML, replacing the Privly link with the cleartext data. Privly does not support the server-side computation that is enabled by Sieve.

ABE-protected storage: Persona [10], Priv.io [73], and Cachet [52] use ABE to selectively expose encrypted user data. In Persona and Priv.io, each user keeps her data in private cloud storage; in Cachet, data is stored in a peer-to-peer, distributed hash table. Unlike Sieve, these systems cannot delegate access to arbitrary third party services. Persona, Priv.io, and Cachet also trust each device for the lifetime of the system, whereas Sieve can recover from the loss of individual devices. Finally, Sieve provides a concrete revocation protocol that safeguards user data if storage servers are compromised. Priv.io has no revocation strategy, and Persona suggests re-keying data, but does not provide a specific mechanism. Cachet does implement revocation, but requires a trusted proxy which must interpose on all decryption operations, even in the common case that revocation is not underway [36]. Cachet's revocation scheme also does not re-encrypt data on storage providers; thus, objects that are encrypted with revoked keys are vulnerable to subsequent compromises of the storage provider.

Predicate encrypted storage: GORAM [46] allows users to selectively share their cloud data with other users. Clients place encrypted data on servers so that servers cannot inspect it, and clients hide their access patterns from servers using ORAM shuffling techniques [44]. Like Sieve, GORAM tags data objects with attributes; unlike Sieve, GORAM uses attribute-hiding predicate encryption [38, 61] to prevent storage servers from learning attribute values.

GORAM's use of oblivious RAM and predicate encryption provides stronger security than Sieve, but there is a performance cost. To hide data access patterns from storage servers, GORAM clients must perform $O(\text{polylog}(n))$ additional accesses. Hiding attribute values using predicate encryption substantially increases GORAM's ciphertext size, and slows both encryption and decryption.

GORAM is also less user-friendly than Sieve. For example, GORAM forces users to determine a priori the maximum number of principals that can be mentioned in access control lists; if this list changes, a user must reinitialize her database. GORAM also has no revocation scheme, and no protocol to recover from lost user devices.

Access delegation schemes: OAuth [37] is a widely used protocol for sharing cloud data across different web services. OAuth policies are written by web services, not by users, so users lack true authority over their access controls. OAuth also does not leverage cryptography to protect user storage or enforce access policies. As a result, users have no strong assurances about how their data is exposed. OAuth is also vulnerable to various kinds of data leaks [33, 64]. AAuth [65] is an extension of OAuth which uses cryptography to delegate access to encrypted data. However, AAuth relies on the existence of various trusted parties to enforce access policies. In Sieve, users generate their own policies and distrust the storage server and third party applications. Sieve's policy language is also richer than AAuth's fixed policy schemas.

The OAuth protocol generates a token that principals use to access sensitive data. Web services define many other types of "bearer tokens." HTTP cookies [11] are a classic example. Macaroons [14] improve upon cookies, using chained HMACs to verify and attenuate capabilities as a macaroon is passed between multiple parties. Cookies and macaroons vouch for a principal's post-authorization status, whereas Sieve deals with the authorization itself.

7 Conclusions

Sieve is a new access control system that allows users to selectively expose their private cloud data to third party web services. Sieve uses attribute-based encryption to translate human-understandable access policies into cryptographically enforceable restrictions. Unlike prior solutions for encrypted storage, Sieve is compatible with rich, legacy web applications that require server-side computation. Sieve is also the first ABE system that protects against device loss and supports full revocation of both data and metadata. As a proof of concept, we integrated Sieve with two open-source web services, demonstrating that Sieve is a practical approach for restricting access to sensitive user data.

Acknowledgments

We thank our anonymous reviewers and our shepherd Brad Karp for their useful feedback. This work was partially supported by an NSF Graduate Research Fellowship (Grant No. 2013135952) and by NSF awards CNS-1053143 and CNS-1413920.

References

[1] Instagram. http://www.instagram.com.

[2] Lark. http://www.web.lark.com.

[3] Piwigo. http://piwigo.org/.

[4] Reddit. https://www.reddit.com.

[5] Share priv(ate).ly. https://priv.ly.

[6] Stack overflow. http://www.stackoverflow.com.

[7] F. Aloul, S. Zahidi, and W. El-Hajj. Two factor authentication using mobile phones. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, pages 641–644, 2009.

[8] Amazon. Put object. http://docs.aws.amazon.com/AmazonS3/latest/API/RESTObjectPUT.html.

[9] S. D. Applegate. Social engineering: Hacking the wetware! *Information Security Journal: A Global Perspective*, 18(1):40–46, Jan. 2009.

[10] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: an online social network with user-defined privacy. *ACM SIGCOMM Computer Communication Review*, 39(4):135–146, 2009.

[11] A. Barth. HTTP state management mechanism. RFC 6265, Internet Engineering Task Force, Apr. 2011.

[12] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 478–492, San Francisco, CA, May 2013.

[13] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage*, 9(4):12, 2013.

[14] A. Birgisson, J. G. Politz, U. Erlingsson, A. Taly, M. Vrable, and M. Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2014.

[15] D. Boneh, K. Lewi, H. Montgomery, and A. Raghunathan. Key homomorphic PRFs and their applications. In *Proceedings of the 33rd Annual International Cryptology Conference (CRYPTO)*, pages 410–428. Santa Barbara, CA, Aug. 2013.

[16] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges. In *Proceedings of the 8th IACR Theory of Cryptography Conference (TCC)*, pages 253–273, Providence, RI, Mar. 2011.

[17] D. Brickley and L. Miller. FOAF vocabulary specification 0.99. http://xmlns.com/foaf/spec/, Jan. 2014.

[18] J. Camenisch and A. Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Proceedings of the 20th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 93–118, Innsbruck, Austria, May 2001.

[19] T. Chajed, J. Gjengset, J. van den Hooff, M. F. Kaashoek, J. Mickens, R. Morris, and N. Zeldovich. Amber: Decoupling user data from web applications. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[20] R. Chandra, P. Gupta, and N. Zeldovich. Separating web applications from user data storage with BStore. In *Proceedings of the USENIX Conference on Web Application Development*, pages 1–14, Boston, MA, June 2010.

[21] R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 concepts and abstract syntax. https://www.w3.org/TR/rdf11-concepts/, Feb. 2014.

[22] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol. RFC 5246, Network Working Group, Aug. 2008.

[23] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Usenix Security Symposium*, pages 303–320, San Diego, CA, Aug. 2004.

[24] C. Dwork. Differential privacy. In *Encyclopedia of Cryptography and Security*, pages 338–340. Springer, 2011.

[25] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[26] S. Gaw and E. W. Felten. Password management strategies for online accounts. In *Proceedings of the*

2nd Symposium On Usable Privacy and Security, pages 44–55, Pittsburgh, PA, July 2006.

[27] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[28] D. K. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 16–25, Pacific Grove, CA, Oct. 1991.

[29] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 89–98, Alexandria, VA, Oct.–Nov. 2006.

[30] M. Green, A. Akinyele, and M. Rushanan. libfenc: The functional encryption library. `https://code.google.com/p/libfenc/`.

[31] M. Hamburg. Ed448-goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, June 2015. `http://eprint.iacr.org/`.

[32] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song. ShadowCrypt: Encrypted web applications for everyone. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 1028–1039, Scottsdale, AZ, Nov. 2014.

[33] E. Homakov. How we hacked Facebook with OAuth2 and Chrome bugs, February 2013. `http://homakov.blogspot.com/2013/02/hacking-facebook-with-oauth2-and-chrome.html`.

[34] Intuit. Mint. `http://www.mint.com`.

[35] J. Ioannidis, S. Ioannidis, A. Keromytis, and V. Prevelakis. Fileteller: Paying and getting paid for file storage. In *Proceedings of the 6th International Financial Cryptography Conference*, pages 282–299, Southampton, Bermuda, Mar. 2002.

[36] S. Jahid, P. Mittal, and N. Borisov. EASiER: Encryption-based access control in social networks with efficient revocation. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 411–415, Hong Kong, Mar. 2011.

[37] M. Jones and D. Hardt. The OAuth 2.0 authorization framework: Bearer token usage. RFC 6750, Internet Engineering Task Force, Oct. 2012.

[38] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Proceedings of the 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 146–162. Istanbul, Turkey, Apr. 2008.

[39] J. Leskovec, N. Milic-Frayling, M. Grobelnik, and J. Leskovec. Extracting summary sentences based on the document semantic graph. Technical Report MSR-TR-2005-07, Microsoft Research, Jan. 2005.

[40] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–106, San Francisco, CA, Dec. 2004.

[41] Y. Lindell and B. Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[42] H. Lipmaa, P. Rogaway, and D. Wagner. Ctr-mode encryption. In *Proceedings of the 1st NIST Workshop on Modes of Operation*, Baltimore, MD, Oct. 2000.

[43] D. Litzenberger. PyCrypto: The Python cryptography toolkit, June 2014. `https://www.dlitz.net/software/pycrypto/`.

[44] J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman. Shroud: ensuring private access to large-scale data in the data center. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 199–213, San Jose, CA, Feb. 2013.

[45] B. Lynn. *On the implementation of pairing-based cryptosystems*. PhD thesis, Stanford University, 2007.

[46] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder. Privacy and access control for outsourced personal records. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2015.

[47] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[48] A. Miyaji, M. Nakabayashi, and S. Takano. New explicit conditions of elliptic curve traces for FR-reduction. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E84-A(5):1234–1243, 2001.

[49] MongoDB. MongoDB. `https://www.mongodb.org/`.

[50] M. Naor, B. Pinkas, and O. Reingold. Distributed pseudo-random functions and KDCs. In *Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 327–346, Prague,

Czech Republic, May 1999.

[51] I. Neamtiu, J. Bardin, M. R. Uddin, D.-Y. Lin, and P. Bhattacharya. Improving cloud availability with on-the-fly schema updates. In *Proceedings of the 19th International Conference on Management of Data*, pages 24–34, 2013.

[52] S. Nilizadeh, S. Jahid, P. Mittal, N. Borisov, and A. Kapadia. Cachet: A decentralized architecture for privacy-preserving social networking with caching. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 337–348, Nice, France, Dec. 2012.

[53] Oracle. BerkeleyDB. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`.

[54] OWASP. Cross-site scripting (XSS). `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`.

[55] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX Annual Technical Conference*, Portland, OR, June 2011.

[56] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, Cascais, Portugal, Oct. 2011.

[57] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using Mylar. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 157–172, Seattle, WA, Apr. 2014.

[58] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek. Online, asynchronous schema change in F1. *Proceedings of the VLDB Endowment*, 6(11):1045–1056, 2013.

[59] E. A. Rundensteiner, A. Koeller, and X. Zhang. Maintaining data warehouses over changing information sources. *Communications of the ACM*, 43(6):57–62, 2000.

[60] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[61] E. Shen, E. Shi, and B. Waters. Predicate privacy in encryption systems. In *Proceedings of the 6th IACR Theory of Cryptography Conference (TCC)*, pages 457–473, San Francisco, CA, Mar. 2009.

[62] R. P. Singh, C. Shen, A. Phanishayee, A. Kansal, and R. Mahajan. A case for ending monolithic apps for connected devices. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[63] C. A. Soules and G. R. Ganger. Toward automatic context-based attribute assignment for semantic file systems. Technical Report CMU-PDL-04-105, Parallel Data Laboratory, Carnegie Mellon University, June 2004.

[64] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 378–390, Raleigh, NC, Oct. 2012.

[65] A. Tassanaviboon and G. Gong. OAuth and ABE based authorization in semi-trusted cloud computing: AAuth. In *Proceedings of the 2nd International Workshop on Data Intensive Computing in the Clouds*, pages 41–50, 2011.

[66] Tides Center. Open mHealth. `http://www.openmhealth.org/`.

[67] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb.–Mar. 2010.

[68] M. S. Turan, E. Barker, W. Burr, and L. Chen. Recommendation for password-based key derivation. Technical Report SP 800-132, NIST, Dec. 2010.

[69] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 473–485, Lombard, IL, Apr. 2013.

[70] Y. Wang, K. Streff, and S. Raman. Smartphone security challenges. *IEEE Computer*, 45(12):52–58, Dec. 2012.

[71] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 329–341, Lombard, IL, Apr. 2013.

[72] Yahoo. Flickr. `https://flickr.com`.

[73] L. Zhang and A. Mislove. Building confederated web-based services with Priv.io. In *Proceedings of the 1st ACM Conference on Online Social Networks*, pages 189–200, 2013.

Earp: Principled Storage, Sharing, and Protection for Mobile Apps

Yuanzhong Xu Tyler Hunt Youngjin Kwon Martin Georgiev
Vitaly Shmatikov[†] Emmett Witchel

The University of Texas at Austin [†]*Cornell Tech*

Abstract

Modern mobile apps need to store and share structured data, but the coarse-grained access-control mechanisms in existing mobile operating systems are inadequate to help apps express and enforce their protection requirements.

We design, implement, and evaluate a prototype of Earp, a new mobile platform that uses the relational model as the unified OS-level abstraction for both storage and inter-app services. Earp provides apps with structure-aware, OS-enforced access control, bringing order and protection to the Wild West of mobile data management.

1 Introduction

Modern mobile apps communicate and exchange data with other apps almost as much as they communicate and exchange data with the operating system. Many popular apps now occupy essential places in the app "ecosystem" and provide other apps with services, such as storage, that have traditionally been the responsibility of the OS. For example, an app may rely on Facebook to authenticate users, Google Drive to store users' data, WhatsApp to send messages to other users, Twitter to publicly announce users' activities, etc.

Traditionally, operating systems have provided abstractions and protection for storing and sharing data. The data model in UNIX [34] is byte streams, stored in files protected by owner ID and permission bits and accessed via file descriptors. UNIX has a uniform access-control model for both storage and inter-process communication: users specify permissions on files, pipes, and sockets, and the OS dynamically enforces these permissions.

Modern mobile platforms provide higher-level abstractions to manage *structured data*, and relational databases have become the de facto hubs for apps' internal data [40]. These abstractions, however, are realized as app-level libraries. Platform-level access control in Android and iOS inherits UNIX's coarse-grained model and has no visibility into the structure of apps' data. Today, access control in mobile platforms is a mixture of basic UNIX-style mechanisms and ad hoc user-level checks spread throughout different system utilities and inter-app services. Apps present differing APIs with ad hoc access-control semantics, different from those presented by the OS or other apps. This leaves apps without a clear and consistent model for managing and protecting access to users' data and leads to serious security and privacy vulnerabilities (see §2).

In this paper, we explore the benefits and challenges of using the relational model as the unified, platform-level abstraction of structured data. We design, implement, and evaluate a prototype of Earp, a new mobile platform that uses this model for both storage and inter-app services, and demonstrate that it provides a principled, expressive, and efficient foundation for the data storage, data sharing, and data protection needs of modern mobile apps.

Our contributions. First, we demonstrate how apps can use the relational model not just to define data objects and relationships, but also to *specify access rights directly as part of the data model*. For example, an album may contain multiple photos, each of which has textual tags; the right to access an album confers the right to access every photo in it and, indirectly, all tags of these photos.

Second, we propose a *uniform, secure data-access abstraction* and a new kind of reference monitor that has visibility into the structure of apps' data and can thus enforce fine-grained, app-defined access-control policies. This enables apps to adhere to the principle of least privilege [36] and expose some, but not all, of users' private data to other apps. App developers are thus relieved of the responsibility for writing error-prone access-control code. The unifying data-access abstraction in Earp is a *subset descriptor*. Subset descriptors are capability-like handles that enable the holder to operate on some rows and columns of a database, subject to restrictions defined by the data owner. Our design preserves efficiency of both querying and access control.

Third, we *implement and evaluate a prototype of Earp* based on Firefox OS, a browser-based mobile platform where all apps are written in Web languages such as HTML5 and JavaScript. Apps access data and system resources via the trusted browser runtime, which acts as the OS from the app's viewpoint. The browser-based design enables Earp to conveniently add its data abstractions and access-control protections to the platform layer while maintaining support for legacy APIs.

Fourth, to demonstrate how apps benefit from Earp's structured access control, we adapt or convert several *essential utilities and apps*. We show how local apps, such as the photo manager, contacts manager, and email client, can use Earp to impose fine-grained restrictions on other apps' access to their data—for example, elide sensitive data fields, support private photos and albums, filter contacts based on categories, or temporarily grant access to an attachment file. We also show how remote services, such as Google Drive and an Elgg-based social-networking service, can implement local proxy apps that use Earp to securely share data with other apps without relying on protocols like OAuth.

We hope that by providing efficient, easy-to-use storage, sharing, and protection mechanisms for structured data, Earp raises the standards that app developers expect from their mobile platforms and delivers frontier justice to the insecure, ad hoc data management practices that plague existing mobile apps.

2 Inadequacy of existing platforms

In today's mobile ecosystem, many apps act as data "hubs." They store users' data such as photos and contacts, make this data available to other apps, and protect it from unauthorized access. The data in question is often quite complex, involving multiple, inter-related objects—for example, a photo gallery is a collection of photos, each of which is tagged with user's notes.

Inadequate protection for storage. Existing platforms do not provide adequate support for mobile apps' data management. Without system abstractions for storing and protecting data, app developers roll their own and predictably end up compromising users' privacy. For example, Dropbox on Android stores all files in public external storage, giving up all protection. WhatsApp on iOS automatically saves received photos to the system's gallery. When the email app on Firefox OS invokes a document viewer to open an attachment, the attachment is copied to the SD card shared by all apps.

A systematic study [54] in 2013 discovered 2,150 Android apps that unintentionally make users' data—SMS messages, private contacts, browsing history and bookmarks, call logs, and private information in instant messaging and social apps (e.g., the most popular Chinese social network, Sina Weibo)—available to any other app.

Inadequate protection for inter-app services. Services and protocols that involve multiple apps have suffered from serious security vulnerabilities and logic bugs [27, 44, 48, 49, 51]. While vulnerabilities in individual apps can be patched, the root cause of this sorry state of affairs is the inadequacy of the protection mechanisms on the existing mobile platforms, which cannot support the principle of least privilege [36].

Existing platforms provide limited facilities for sharing data via inter-app services. Android apps can use *content providers* to define background data-sharing services with a database-like API, where data are located via URIs. Android's reference monitor enforces only coarse-grained access control for content providers based on static permissions specified in app manifests [2]. Even though permissions can be specified for particular URI paths, they can only be used for static, coarse categories (e.g., images or audio in Media Content Provider) because it is impossible to assign different permissions to dynamically created objects, nor enforce custom policies for different client apps. If a service app needs fine-grained protection, writing the appropriate code is entirely the app developer's responsibility. Unsurprisingly, access control for Android apps is often broken [38, 54].

Android also has a URI permission mechanism [1] for fine-grained, temporary access granting. The access-control logic still resides in the application itself, making URI permissions difficult to use for programmatic access control. Android mostly uses them to involve the user in access-control decisions, e.g., when the user clicks on a document and chooses an app to receive it.

In iOS, apps cannot directly share data via the file system or background services. For example, to share a photo, apps either copy it to the system's gallery, or use app extensions [24] which require user involvement (e.g., using a file picker) for every operation.

Without principled client-side mechanisms for protected sharing, mobile developers rely on server-side authentication protocols such as OAuth that give third-party apps restricted access to remote resources. For example, Google issues OAuth tokens with restricted access rights, and any app that needs storage on Google Drive attaches these tokens to its requests to Google's servers [18, 19]. Management of OAuth tokens is notoriously difficult and many apps badly mishandle them [48], leaving these apps vulnerable to impersonation and session hijacking due to token theft, as well as identity misbinding and session swapping attacks such as cross-site login request forgery [44]. In 2015, a bug in Facebook's OAuth protocol allowed third-party apps to access users' private photos stored on Facebook's servers [14].

Inadequate protection model. Protection mechanisms on the existing platforms are based on permissions attached to individual data objects. These objects are typically coarse-grained, e.g., files. Even fine-grained permissions (e.g., per-row access control lists in a database) do not support the protection requirements of modern mobile apps. The fundamental problem is that data objects used by these apps are *inter-related*, thus any inconsistency in permissions breaks the semantics of the data model.

Per-object permissions fail to support even simple, common data sharing patterns in mobile apps. Consider a photo collection where an individual photo can be accessed directly via the camera roll interface, or via any album that includes this photo. As soon as the user wants to share an album with another app, the per-object permissions must be changed for every single photo in the album. Since other types of data may be related to photos (e.g., text tags), the object-based permission system must compute the transitive closure of reachable objects in order to update their permissions. This is a challenge for performance and correctness.

In practice, writing permission management code is complex and error-prone. App developers thus tend to choose coarse-grained protection, which does not allow them to express, let alone enforce their desired policies.

3 Design goals and overview

Throughout the design of Earp, we rely on the platform (i.e., the mobile OS) to protect the data from unauthorized access and to confine non-cooperative apps. Earp provides several platform-enforced mechanisms and abstractions to make data storage, sharing, and protection in mobile apps simpler and more robust.

- Apps in Earp store and manage data using a uniform, relational model that can easily express relationships between objects as well as access rights. This allows app developers to employ standard database abstractions and relieves them of the need to implement their own data management.
- Apps in Earp give other apps access to the data via structured, fine-grained, system-provided abstractions. This relieves app developers of the need to implement ad hoc data-access APIs.
- Apps in Earp rely on the platform to enforce their access-control policies. This separation of policy and mechanism relieves app developers of the need to implement error-prone access-control code.

Efficient system-level enforcement requires the platform to have visibility into the data structures used by apps to store and share data. In the rest of the paper, we describe how this is achieved in Earp.

3.1 Data model

UNIX has a principled approach for protecting both storage and IPC channels, based on a unifying API—file descriptors. On modern mobile platforms, however, data management has moved away from files to structured storage such as databases and key/value stores.

In Earp, the unifying abstraction for both storage and inter-app services is *relational data*. This approach (1) helps express relationships between objects, (2) integrates access control with the data model, and (3) provides a uniform API for data access, whether by the app that owns the data or by other apps.

Unifying storage and services is feasible because Earp apps access inter-app services by reading and writing structured, inter-related data objects via relational APIs that are similar to those of storage. A service is defined by four *service callbacks* (§5), which Earp uses as the primitives to realize the relational API.

Earp uses the same protection mechanism for remote resources. For example, a remote service such as Google Drive can have a *local proxy* app installed on the user's device, which defines an inter-app service that acts as the gateway for other apps to access Google's remote resources. Earp enforces access control on the proxy service in the same way as it does with all inter-app services, avoiding the need for protocols such as OAuth.

Earp not only makes it easier to manage structured data that is pervasive in mobile apps, but also maintains efficient, protected access to files and directories. Earp uses files and directories internally, thus avoiding the historical performance problems of implementing a file system on top of a database [50].

3.2 Access rights

All databases and services in Earp have an owner app. The owner has the authority to define policies that govern other apps' access, making Earp a discretionary access control system. The names of databases and services are unique and prefixed by the name of the owner app.

Earp's protection is fine-grained and captures the relationships among objects. In the photo gallery example, each photo is associated with some textual tags, and photos can be included in zero, one, or several albums. Fine granularity is achieved by simple *per-row ACLs*, allowing individual photos to each have different permissions. However, per-object permissions alone can create performance and correctness problems when apps share collections of objects (§2).

To enable efficient and expressive fine-grained permissions for inter-related objects, Earp introduces **capability relationships**—relationships that confer access rights among related data. For example, if an app that has access rights to an album traverses the album's capability relationship to a photo, the app needs to automatically obtain access rights to this photo, too. Capability relationships only confer access rights when traversed in one direction. For example, having access to a photo does not grant access to all albums that include this photo.

Capability relationships make it easy for apps to share ad hoc collections. For example, the photo gallery can create an album for an ephemeral messaging app like Snapchat, enabling the user to follow the principle of least privilege and install Snapchat with permissions to access only this album (and, transitively, all photos in this album and their tags).

Capability relationships also enable Earp to use very simple ACLs without sacrificing the expressiveness of access control. There are no first-class concepts like groups or roles, but they can be easily realized as certain capability relationships.

3.3 Data-access APIs

In Earp, access to data is performed via **subset descriptors**. A subset descriptor is a capability "handle" used by apps to operate on a database or service. The capability defines the policy that mediates access to the underlying structured data, allowing only restricted operations on a subset of this data.

The holder of a subset descriptor may transfer it to other apps, possibly *downgrading* it beforehand (removing some of the access rights). Intuitively, a subset descriptor is a "lens" through which the holder accesses a particular database or service.

Critically, the OS reference monitor ensures that all accesses comply with the policy associated with a given descriptor. Therefore, app developers are only responsible for defining the access-control policy for their apps' data but not for implementing the enforcement code.

Capability relationships make access rights for one object dependent on other objects. This is a challenge for efficiency because transitively computing access-control decisions would be expensive. To address this problem, apps can *create subset descriptors on demand to buffer access-control decisions for future tasks*. For example, an app can use a descriptor to perform joins (as opposed to traversal) to find all photos with a certain tag, then create another descriptor to edit a specific photo based on the result of a previous join. The photo access rights are computed once and bound to the descriptor upon its creation. Earp thus enjoys the benefits of both the relational representation (efficient joins) and the graph representation (navigating a collection to enumerate its members).

To facilitate programming with structured data, Earp provides a library that presents an *object graph* API backed by databases or inter-app services (see an example in Figure 1). This API is functionally similar to the Core Data API in iOS, but each internal node is mapped to a platform-level data object under Earp's protection. This API relieves developers of the need to explicitly handle descriptors or deal with the relational semantics of the underlying data.

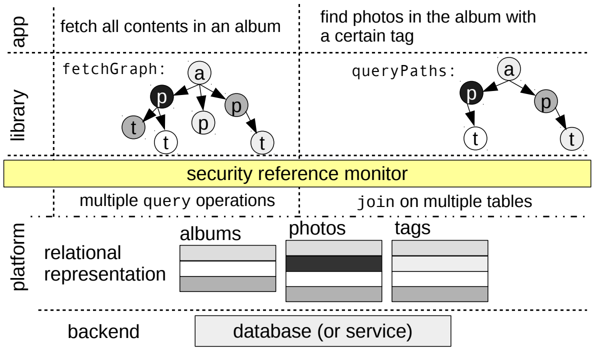

app
fetch all contents in an album
find photos in the album with a certain tag
library
fetchGraph:
queryPaths:
a
p
t
security reference monitor
multiple query operations
join on multiple tables
platform
relational representation
albums
photos
tags
backend
database (or service)

Figure 1: Platform- and library-level representations of structured data in Earp.

3.4 Choosing the platform

Mobile apps are often written in portable Web languages such as HTML5 and JavaScript [46, 47]. Browser-based mobile/Web platforms (e.g., Firefox OS, Chrome, and universal Windows apps) support this programming model by exposing high-level resource abstractions such as "contacts" and "photo gallery" to Web apps, as well as generic structured storage like IndexedDB; they are implemented in a customized, UI-less browser runtime, instead of app-level libraries. All resource accesses by apps are mediated by the browser runtime, although it only enforces all-or-nothing access control.

For our Earp prototype, we chose a browser-based platform, Firefox OS, allowing us to easily add fine-grained protection to many new and legacy APIs. Earp also retains coarse-grained protection on other legacy APIs (e.g., raw files), allowing us to demonstrate Earp's power and flexibility with substantial apps (§7.1).

It is possible to adapt Earp to a conventional mobile platform like Android. For storage, we could port SQLite into the kernel and add access-control enforcement to system calls; alternatively, we could create dedicated system services to mediate database accesses and enforce access-control policies. Non-cooperative apps would be confined by the reference monitor in either the kernel, or the services. For content providers, we could modify the reference monitor to support capability relationships, and require apps to provide unforgeable handles that are similar to subset descriptors when they access data in content providers.

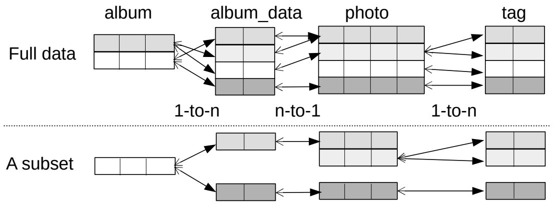


Figure 2: A relational representation of structured data. We show the entire data set and a subset chosen by a combination of row and column filtering. Relationships across tables are always bidirectional, but capability relationships are unidirectional as indicated by solid arrows.

4 Data storage and protection

UNIX stores byte streams in files protected by owner ID and permission bits and accessed via file descriptors. Earp stores structured data in relational databases protected by permission policies and accessed via *subset descriptors*. Because structured data is more complex than byte streams, Earp must provide more sophisticated protection mechanisms than what is needed for files. Before describing these mechanisms, we give a brief overview of the relational data model and how it's used in Earp.

4.1 Data model

Earp represents structured data using a relational model. The same relational API is used for storage and inter-app services (§5). The back end of this API can be, respectively, a database or a service provided by another app.

Each data object in Earp is a *row* in some *table*, as shown in Figure 2. An object in one table can have relationships with objects in other tables. For example, a photo object is a row in the photo table with a column for raw image data, several columns for EXIF data (standard metadata such as the location where the photo was taken), and a relationship with the tag table, where tags store textual notes. Storing tags in a separate table allows photos to have an arbitrary number of tags that can be queried individually. Relationships in Earp are standard database relationships, as summarized below, but the concept of a capability relationship (§4.2) is a new contribution and the cornerstone of efficient access control in Earp.

Relationships have different cardinalities. For example, the relationship between a photo and its tags is *1-to-n* from the photo to its tags, or, equivalently, *n-to-1* from the tags to the photo. *1-to-1*, or, more precisely *(1|0)-to-1*, is a special case of n-to-1. For example, each digital camera has a single product profile which may or may not be present in the photo's EXIF.

Logically, the relationship between albums and photos is *n-to-n*, because a photo can be included in multiple albums and an album can contain multiple photos. Like many relational stores, Earp realizes n-to-n relationships by adding an intermediate table. In our example, we call the intermediate table *album_data*. The album-album_data relationship is 1-to-n, and the album_data-photo relationship is n-to-1. All four tables are illustrated in Figure 2.

4.2 Access rights

Access control lists. Each database in Earp is owned by a single app. Rows have very simple access control lists (ACLs) to control their visibility to other apps. Each row is either public, or private to a certain app. If a table does not have an AppId column, it can be directly accessed only by the owner of the database. If an Earp table has an AppId column, its value encodes the ACL: zero means that the row is public, positive *n* means that the row is private to the app whose ID is *n*. Any app can read or write public rows. Without an appropriate capability relationship (see below), apps can only read or write their own private rows.

Relationships create challenges for ACLs because they are traversed at run time and their transitive closure may include many objects. If ACLs were the only protection mechanism, an app that wants to share a photo with another app would have to modify the ACLs for all tags—either by making each ACL a list containing both apps, or by creating a group.

Capability relationships. A relationship is logically bidirectional. For example, given a photo, it is possible to retrieve its tags, and given a tag, it is possible to retrieve the photo to which it is attached. In Earp, however, only a single direction can confer access rights, as specified in the schema definition. These *capability relationships* are denoted as solid arrows in Figure 2.

We use x$\xleftrightarrow{1:n}$y to denote a 1-to-n capability relationship between tables x and y, which confers access rights when moving from the 1-side (x) to the n-side (y). Similarly, x$\xleftrightarrow{n:1}$y denotes an n-to-1 capability relationship that confers access when moving from the n-side to the 1-side. x$\xleftrightarrow{n:1}$y denotes a non-capability relationship that does not confer access rights.

In the photo gallery example,

• photo$\xleftrightarrow{1:n}$tag. Having a reference to a photo grants the holder the right to access all of that photo's tags, but not the other way around. Therefore, if an app asks for all photos with a certain tag, it will receive only the matching photos that are already accessible to it (via ownership, ACL, or capability relationship).

• album$\xleftrightarrow{1:n}$album_data$\xleftrightarrow{n:1}$photo. The intermediate table album_data realizes an n-to-n relationship with capability direction from album to photo. Having access to an album thus confers access to the related objects in album_data and photo.

album_data and tag are both on the n-side of some x$\xrightarrow{1:n}$y relationship, and they are intended to be accessed only via capability relationships. For example, each tag is attached to a single photo and is useful only if the photo is accessible. Typically, such tables do not need ACLs.

We have not needed bidirectional capability relationships in Earp, and they would create cycles that make the access-control model confusing. Therefore, we decided not to support bidirectional capability relationships at the platform level. Earp prevents capabilities from forming cycles, ensuring that the transitive closure of all capability relationships is a directed acyclic graph (DAG).

Groups. A group can be created in Earp by defining a table with an appropriate schema. For example, to support albums that are shared by a group of apps, the app can define another table album_access, with album_access$\xrightarrow{n:1}$album. Each row in album_access is owned by one app and confers access to an album. With this table, even if an album is private to a certain app, it can be shared with other apps via entries in album_access.

Primary and foreign keys. Earp requires that all tables have immutable, non-reusable primary keys generated by the platform. The schema can also define additional keys. Therefore, the $(database, table, primary\_key)$ tuple uniquely identifies a database row.

Cross-table relationships are represented via foreign keys in relational databases. A foreign key specifies an n-to-1 relationship: the table that contains the foreign key column is on the n-side, the referenced table is on the 1-side. If the foreign key column is declared with the UNIQUE constraint, the relationship is (1|0)-to-1.

Earp enforces that a foreign key references the primary key of another table and must guarantee *referential integrity* when the referenced row is deleted [41].

For x$\xrightarrow{1:n}$y where y does not have ACLs, when the referenced row (e.g., a photo) is deleted, the referencing rows (e.g., tags) will be deleted as well, because they are inaccessible and the deleting app has the (transitive) right to delete them.

For other types of relationships, when the referenced row (e.g., a photo) is deleted, Earp by default sets the foreign keys of the referencing rows (e.g., rows in album_data) to NULL. If these rows no longer contain useful data without the foreign key, the schema can explicitly prescribe that they should be deleted. For album_data, it is reasonable to delete the rows because they are merely intermediate relations between albums and photos.

4.3 App-defined access policies

ACLs and capability relationships are generic and enforced by Earp once the schema of a database or service is defined. To enable more expressive access control tailored for relational data, Earp also lets apps define schema-level *permission policies* on their databases and services. These policies govern other apps' access to the data.

A policy defines the following for each table:

1. AppID and default insert mode.

2. Permitted operations: insert, query, update, and/or delete.

3. A set of accessible columns (projection).

4. A set of columns with fixed values on insert/update.

5. A set of accessible rows (selected by a WHERE clause, in addition to ACL-based filtering).

The AppID is a number that identifies the controlling app as the basis for ACLs, much like the user ID identifies the user as the basis for interpreting file permission bits. The default insert mode indicates if data inserted into the database is public or private to the inserting app.

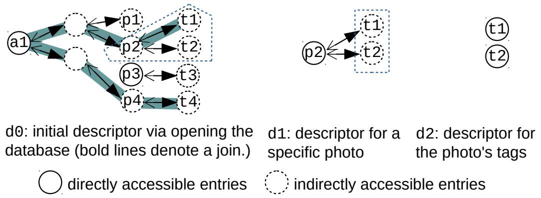
Data access in Earp is expressed by four SQL operations—insert, query, update, and delete—inspired by Android's SQLite API (omitting administrative functions like creating tables). Read-only access is realized by restricting the available SQL operations to query only. Control over writing is fine-grained: for example, an app can limit a client of the API to only insert into the database, without giving it the ability to modify existing entries.

The permission policy can filter out certain rows (e.g., private photos) and columns (e.g., phone numbers of contacts), making them "invisible" to the client app. In addition, values of certain columns can be fixed on insert/update. For example, a Google Drive app can enforce that apps create files only in directories named by their official identifiers.

Just like the owner ID and permission bits of a file constrain the file descriptor obtained by a user when opening a file in UNIX, the permission policy constrains the subset descriptor (see below) obtained by a user when opening a database. While permission bits specify a policy for all users using coarse categories (owner, group, others), Earp lets apps specify initial permission policies for individual AppIDs, as well as the default policy. Figure 6 in Section 7.1 shows examples of policy definitions.

4.4 Data-access APIs

Earp provides two levels of APIs to access relational data: direct access via subset descriptors and object-graph access via a library.


a1
p1
p2
p3
p4
t1
t2
t3
t4
d0: initial descriptor via opening the database (bold lines denote a join.)
d1: descriptor for a specific photo
d2: descriptor for the photo's tags
directly accessible entries
indirectly accessible entries

```
var d0 = navigator.openDB('sys/gallery');
var cursor = d0.joinTransClosure(['album','album_data',
      'photo', 'tag'], where); // join
cursor.onsuccess = function(event) {
  ... // navigate to a row (the first bold line above)
  // d1: descriptor for photo in cursor's current row
  var d1 = cursor.getSelfDesc('photo');
  // d2: descriptor for the current photo's tags
  var d2 = cursor.getRefDesc('photo', 'tag');
}
```

Figure 3: A database join using an initial subset descriptor, then creating new descriptors to represent subsets of the result. The figure includes a visual depiction of the data accessible from the different descriptors.

4.4.1 Subset descriptors

Apps in Earp access databases and services via subset descriptors. When an app opens a database or service that it owns, it obtains a full-privilege descriptor. If it opens another app's database or service, it obtains a descriptor with the owner's (default or per-app) permission policy.

Subset descriptors are created and maintained by Earp; apps manipulate opaque references to descriptors. Therefore, Earp initializes descriptors in accordance with the database owner's permission policy, and apps cannot tamper with the permissions of a descriptor (though descriptors can be downgraded, as discussed below).

Efficiently working with descriptors. An example of working with descriptors is shown in Figure 3. The app receives descriptor `d0` when it opens the database. It can use `d0` to access albums or photos as permitted by their ACLs. The code in Figure 3 will succeed in performing a join using `d0` because Earp verifies that all tables can be reached by traversing the capability relationships from a root table (`album` in this case), and that entries in different tables are related via corresponding foreign keys.

However, using `d0` is not always efficient for all tasks, because access rights on some objects can only be computed transitively. To minimize expensive cross-table checks, an app can create more descriptors that directly encode computed access rights over transitively accessible objects. Once such a descriptor is created, the app can use it to access the corresponding objects without recomputing access rights. In Figure 3, when the app successfully performs a query, join, or insert for a particular photo via `d0`, this proves to Earp that it can access the photo in question. Therefore, Earp lets it obtain a new descriptor `d2`, which allows the app to operate only on the entries in the `tag` table whose foreign key matches the photo's primary key. Access rights are verified and bound to `d2` upon its creation, thus subsequent operations on `d2` are not subject to cross-table checks. Any tag created using the `d2` descriptor will belong to the same photo because `d2` fixes the foreign key value to be the photo's primary key. As discussed in §4.4.2, the object graph library automates creation and management of descriptors.

Transferring and downgrading descriptors. An app can pass its descriptor to another app or it can create a new descriptor based on the one it currently holds (e.g., create `d1` based on `d0` in Figure 3). When a new descriptor is generated based on an existing one, all access restrictions are inherited. For example, if the existing descriptor does not include some columns, the new one will not have those columns, either; if the existing descriptor is query-only, so will be the new one; fixed values for columns, if any, are inherited, too.

When delegating its access rights, an app may create a *downgraded* descriptor. For example, an app that has full access to an album may create a read-and-update descriptor for a single photo before passing it to a photo editor. A downgraded descriptor can also deny access to certain relationships by making the column containing the foreign key inaccessible.

Revoking descriptors. By default, a subset descriptor is valid until closed by the holding app. However, sometimes an app needs more control over a descriptor passed to another app. Therefore, Earp supports *transitive revocation*. When an app explicitly revokes a subset descriptor, all descriptors derived from it will also be revoked, including descriptors that are copied or transferred[1] from it, as well as those generated based on query results. In this way, App A can temporarily grant access to App B by passing a descriptor `d` to it, then revoke App B's copy of `d` (and derived descriptors) afterwards by revoking the original copy in App A itself.

Creating relationships. A foreign key in Earp may imply access rights. For $\mathtt{x}\xleftrightarrow{1:n}\mathtt{y}$, foreign keys are never specified by the app. For example, inserting a tag for a photo can only be done via a descriptor generated for that photo's tags, i.e., `d2` in Figure 3, which fixes the foreign key value. This prevents an app from adding tags to a photo that it cannot access.

For $\mathtt{x}\xleftrightarrow{n:1}\mathtt{y}$, however, the app needs to provide a foreign key when creating a new row in `x`. For example, to add an existing photo to an album, the app needs to add a row in `album_data` with a foreign key referencing the photo. In this case, Earp must ensure that the app has some administrative rights over the referenced photo, because this operation makes the photo accessible to anyone that has access to the album. An analogy is changing file permissions in UNIX via `chmod`, which also requires

[1] Transferring a descriptor generates a new copy of the descriptor in the receiving app. This copy is derived from the original descriptor.

administrative rights (matching UID or root).

To create such a reference, Earp requires an app to specify the foreign key value in the form of an unforgeable token. The app can obtain such a token via a successful insert or query on the referenced row, provided that the row is public or owned by the app. This proves that the app has administrative rights over the row.

4.4.2 Object graph library

As mentioned in Section 3, Earp provides a library that implements an object graph API on top of the relational data representation. Rows (e.g., photos) are represented as JavaScript objects. Related objects (e.g., photos and tags) are attached to each other via object references. The corresponding descriptors are computed and managed internally by the library. As Figure 1 illustrates for our running photo gallery example, an album can be retrieved (or stored) as a graph, and searching for photos with a certain tag can be done via a path query in this graph.

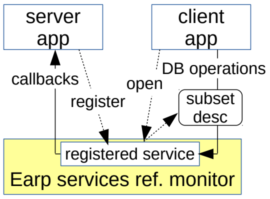
An app can use this library to conveniently construct a subgraph from an entry object that has capability or non-capability relationships with other objects. The lightweight nature of subset descriptors allows the library to proactively create descriptors as the app is performing queries. Internally, the library automates descriptor management and chooses appropriate descriptors for each operation. For example, it has dedicated descriptors for simple function APIs such as `addObjectRef` to create objects that have relationships with existing ones, as well as APIs that facilitate more complex operations, such as:

- `populateGraph`: populate a subgraph from a starting node (e.g., fetch all data from an album);
- `storeGraph`: store objects from a subgraph to multiple tables (e.g., store a new photo along with its tags);
- `queryPaths`: find paths in a subgraph that satisfy a predicate (e.g., find photos with a certain tag in an album).

5 Data sharing via inter-app services

In Earp, sharing non-persistent data between apps relies on the same relational abstractions as storage. In particular, data is accessed through subset descriptors that control which operations are available and which rows and columns are visible (just like for storage). The OS in Earp interposes on inter-app services, presents a relational view of the shared data, and is fully responsible for enforcing access control.

Figure 4 illustrates inter-app services in Earp. The *server app* is the provider of the data, the *client app* is a recipient of the data. In Earp, the server app defines and registers a named service, implemented with four *service callbacks*. To client apps, this service appears as a database with a set of *virtual tables* and clients use subset descriptors to access this "database." Defining virtual tables via callbacks is a standard idea, and a similar mechanism exists in SQLite [42]. Earp uses a subset of this interface tailored for the needs of mobile apps.



Earp translates DB operations into service callbacks

client operation	service callbacks
query	list
insert	add
update	list, alter
delete	list, remove

Figure 4: Inter-app services in Earp.

Virtual tables have the same relational model and are accessed through the same subset descriptors as conventional database tables (§4). The server app can define permission policies on virtual tables, in the same way as for storage databases. Like conventional tables, a virtual table can have a foreign key to another virtual table, defining a capability or non-capability relationship.

5.1 Implementing a relational service API

A service is implemented by defining four service callbacks: `list`, `add`, `alter`, and `remove`. The callbacks operate on *virtual tables* as follows.

- `list`: The server app provides a list of rows in the requested virtual table. This is the only set operation among the four callbacks. The server app also supplies values for the ACL column of any directly accessible table. Many use cases (§7.1), however, only rely on schema-level permission policies, so the server app may simply provide a dummy public value.
- `add`: Given a single row object, the server app adds it to the requested virtual table.
- `alter`: Given a single row object and new values for a set of columns, the server app updates that row in the requested virtual table.
- `remove`: Given a single row object, the server app deletes it from the requested virtual table.

Implementation of the service callbacks is necessarily app-specific. An app can retrieve data in response to a `list` invocation from an in-memory data structure, or fetch it on demand from a remote server via HTTP(S) requests. For example, `list` for the Google Drive service may involve fetching files, while `add` for the Facebook service may result in posting a status update.

5.2 Using a relational service API

Earp interposes on client apps' accesses to a service and converts standard database operations on virtual tables (query, insert, update, delete) into invocations of service callbacks. The reference monitor filters out inaccessible rows and columns and fixes column values according to the subset descriptor held by the client app.

- `query`: Earp invokes `list`, then filters the result set before returning to the client. Multi-table queries (joins) are converted to multiple `list` calls.
- `insert`: Earp sanitizes the client app's input row object by setting the values of fixed columns as specified in the descriptor, then passes the sanitized row to `add`.
- `update`: Earp invokes the `list` callback, performs filtering, sanitizes the new values, then invokes `alter` for each row in the filtered result set. This ensures that only the rows to which the client app has access will be updated, and that the client cannot modify columns that are inaccessible or whose values are fixed.
- `delete`: Earp invokes the `list` callback, performs filtering, then invokes `remove` for each row in the filtered result set.

5.3 Optimizing access-control checks

Earp's strategy of active interposition to enforce access control on inter-app services could reduce performance for certain server implementation patterns. We use several techniques to mitigate the performance impact on important use cases.

Separate data and metadata. Earp's filtering for `list` happens *after* the server app provides the data. Therefore, if the server returns a lot of unstructured "blob" data (e.g, raw image data associated with photos), possibly from a remote host, access control checks could be expensive.

In the common scenario where only metadata columns are used to define selection and access control criteria, the server app can greatly improve performance by separating the metadata and the blob data into two tables. The metadata table is directly visible to the client apps, and Earp performs filtering on it. The blob table is only accessible via a capability relationship (i.e., $\texttt{metadata} \xleftrightarrow{\text{n:1}} \texttt{blob}$). The client app receives the filtered result from the metadata table and can only fetch blobs that are referenced by the metadata rows.

Leverage indexing and query information. Although Earp does not require the server app to check the correctness or security of the data it returns in response to `list`, the server app can significantly reduce the amount of sent data if it already maintains indices on the data and takes advantage of the fact that Earp lets it see the actual client operation that invoked a particular callback.

For example, when a service exports a key/value interface, the server app can learn the requested key from Earp and return only the value for that key. Similarly, if the service acts as a proxy for a local database (e.g., a photo filter for the gallery), Earp sanitizes the client requests based on the client's descriptor and passes the sanitized operations to the service. The service uses Earp's database layer, which has a safe implementation of the relational model.

6 Implementation of Earp

We modified Firefox OS 2.1 to create the Earp prototype. The backend for storage is SQLite, a lightweight relational database that is already used by Firefox OS internally. Firefox OS supports inter-app communication based on a general message passing mechanism. It presents low-level APIs to send and receive JavaScript objects (similar to Android Binder IPC). Earp's inter-app service support is built on top of message passing, but presents higher-level APIs that facilitate access-control enforcement for structured data (similar to Android Content Providers which are built on top of Binder IPC). Our implementation of Earp consists of 7,785 lines of C++ code and 1,472 lines of JavaScript code (counted by CLOC [9]) added to the browser runtime and libraries.

6.1 Storing files

There are two ways to store files in Earp. When per-file metadata (e.g., photo EXIF data and ACLs) is needed, files can be co-located with the metadata in a database with file-type columns. Apps store large, unstructured blob data (e.g., PDF files) using file-type columns, and the only way for them to get handles to these files is by reading from such columns. This eliminates the need for a separate access-control mechanism for files. Internally, Earp stores the blob data in separate files and keeps references to these files in the database. This is a common practice for indexing files, used, for example, in Android's photo manager and email client. Inserting a row containing files is atomic from the app's point of view. This allows Earp to consistently treat data and metadata, e.g., a photo and its EXIF.

If per-file metadata and access control are not needed, an app can store and manage raw files via directory handles. Access control is provided at directory granularity, and apps can have private or shared directories. Internally, Earp reuses the access-control mechanism for database rows to implement per-directory access control, simply by adding a directory-type column which stores directory references. The permissions on a directory are determined by the permissions on the corresponding database row.

6.2 Events and threads

JavaScript is highly asynchronous and relies heavily on events. Therefore, the API of Earp is asynchronous and apps get the results of their requests via callbacks.

Thread pool. Internally, all requests to storage and services are dispatched to a thread pool to avoid blocking the

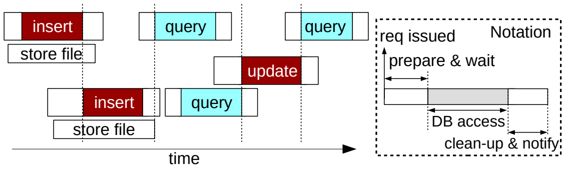


Figure 5: Constraints on request processing order in the thread pool.

app's main thread for UI updates. The thread pool handles all I/O operations for database access and performs result filtering for inter-app services. After completing its processing of a request, Earp dispatches a success or error event to the main thread of the app, which invokes an appropriate callback.

A request may be processed by multiple concurrent threads to maximize parallelism. For example, inserting a row that contains n files will be processed by $n+1$ threads, where the first n threads store the files and the last thread inserts metadata into the database. Although processed concurrently, such an insert request is atomic to apps, because they are not allowed to access the files until the insert finishes. If any thread fails, Earp aborts the operation and removes any written data.

Similarly, a request to a service can also be parallelized. For example, when processing an `update` request, Earp first uses a thread to invoke the `list` callback of the server app and to filter the result; for each row that passes the filter, Earp immediately dispatches an event to invoke the `alter` callback. If `alter` has high latency due to remote access, the server app can also parallelize its processing, e.g., by sending concurrent HTTP(S) requests.

Request ordering. When processing requests, Earp preserves the program order of all write requests (insert, update and delete) and guarantees that apps read (query) their writes. The critical section (database access) of a write waits for all previous requests to complete, while a read waits only for previous writes. Storing blob-type columns, as part of inserts or updates, is parallelized; however, a read must wait for the previous blob stores to complete. Note that an app could request an editable file or directory handle from a database query, but Earp does not enforce the order of reads and writes on the handle. It enforces the order when storing or replacing the whole blob using inserts or updates. Figure 5 shows an example of runtime request ordering.

6.3 Connections and transactions

A subset descriptor is backed by a database connection or a service connection. The program's order of requests is preserved *per connection*. When an app opens a database or a service, Earp creates a new connection for it. Descriptors that are derived from an existing descriptor inherit the same connection. However, the app can also request a new connection for an existing descriptor.

Earp exposes SQLite's support for transactions to apps. An app can group multiple requests in a transaction. If it does not explicitly use the API for transactions, each individual request is considered a transaction. Note that a transaction is for operations on a connection; requests on multiple descriptors could belong to a same transaction if they share the connection. The object graph library uses transactions across descriptors to implement the atomic version of `storeGraph`.

6.4 Safe SQL interface

SQL queries require `WHERE` clauses, but letting apps directly write raw clauses would create an SQL injection vulnerability. Earp uses structured objects to represent `WHERE` clauses and column-value pairs to avoid parsing strings provided by apps and relies on prepared statements to avoid SQL injection.

6.5 Reference monitor

The reference monitor mediates apps' access to data by creating appropriate descriptors for them and enforcing the restrictions encoded in the descriptor when processing apps' requests. Descriptors, requests, and tokens for foreign keys can only be created by the reference monitor; they cannot be forged by apps. They are implemented as native C++ classes with JavaScript bindings so that their internal representation is invisible to apps. These objects are managed by the reference counting and garbage collection mechanisms provided by Firefox OS.

App identity. An app (e.g., Facebook) often consists of local Web code, remote Web code from a trusted origin (e.g., `https://facebook.com`) specified in the app's manifest, and remote Web code from untrusted (e.g., advertising) origins. Earp adopts the app identity model from PowerGate [17], and treats the app's local code and remote code from trusted origins as the same principal, "the app." Web code from other origins is considered untrusted and thus has no access to databases or services.

Policy management. Earp has a global registry of policies for databases and services, specified by their owners. Earp also has a trusted policy manager that can modify policies on any database or service.

7 Evaluation

7.1 Apps

To illustrate how Earp supports sharing and access-control requirements of mobile apps, we implemented several essential apps based on Firefox OS native apps and utilities.

Photo gallery and editor. Gallery++ provides a user interface for organizing photos into albums and applying tags to photos (as in our running example). With the schema shown in Figure 2, Earp automates access control enforcement for Gallery++ and lets it define flexible policies for other apps. For example, when other apps open the photo database, they are granted access to their private photos and albums as well as public photos and albums, but certain fields like EXIF may be excluded.

Gallery++ can also share individual photos or entire albums with other apps (optionally including EXIF and tag information), by passing subset descriptors. For example, we ported a photo editing app called After Effects to Earp but blocked it from directly opening the photo database. Instead, this app can only accept descriptors from Gallery++ when the user explicitly invokes it for the photos she selected in Gallery++. When she finishes editing and returns from After Effects, Gallery++ revokes the descriptor to prevent further access.

Contacts manager. The Earp contacts manager provides an API identical to the Firefox OS contacts manager, thus legacy applications interacting with the manager all continue to work, yet their access is restricted according to the policies imposed by the Earp contacts manager.

The contacts manager stores contacts using seven tables: the main `contact` table in which the columns are simple attributes, five tables to manage attributes that allow multiple entries (e.g., `contact`$\xleftrightarrow{1:n}$`phone` and `contact`$\xleftrightarrow{1:n}$`email`), and the final table that holds contact categories with `category`$\xleftrightarrow{n:1}$`contact`. Categories can be used to restrict apps' access to groups of related contacts. Such a schema enables Earp-enforced custom policies, e.g., a LinkedIn app can be given access only to contacts in the "Work" category, without home address information.

Email. The Firefox OS built-in email client saves attachments to the world-readable device storage (SD card) when it invokes a viewing app to open the attachment.

The Earp email client allows attachments to be exported only to an authorized viewing app, which obtain a subset descriptor to the email app's database. The Earp email client also supports flexible queries from the viewing app, such as "show all pictures received in the past week," or "export all PDF attachments received two days ago".

Elgg social service and client apps. We use Elgg [12], an open-source social networking framework, to demonstrate Earp's support for controlled sharing of app-defined content. We customized Elgg to provide a Facebook-like social service where users can see posts from their friends. There are three components: the Elgg Web server, the Elgg local proxy app, and local client apps. Client apps are not authorized to directly contact the Elgg Web server. Instead, they must communicate with the Elgg local app which defines a service. This service acts as a local proxy and accesses remote resources hosted on the Web server.

```
Activity Map:
 {post: {ops: ['query'],
         cols: ['location']},
  image: {ops: [], cols: []}} // no access
Social Collection:
 {post: {ops: ['query'],
         // WHERE clause (group='public') encoded
         // as a JS object to prevent SQL injection
         rows: {op: '=', group: 'public'}},
  image: {}} // image access implied by post
News:
 {post: {ops: ['insert'],
         fixedCols: [{category: 'news'}]},
  image: {}} // image access implied by post
```

Figure 6: Policies defined for Elgg client apps, represented as JavaScript objects.

A post in Elgg is a text message with associated images. The Elgg app maintains two virtual tables, one for the post text (called `post`), the other for the images (called `image`), with a `post`$\xleftrightarrow{1:n}$`image` relationship.

The service callbacks use asynchronous HTTP requests to fetch data. To optimize bandwidth usage, images are only fetched when the requesting client app has access to the post with which they are associated.

Local access control in Earp provides a simple and secure alternative to OAuth. The Elgg local app defines policies for other apps based on user actions, e.g., via prompts. We implemented several client apps, and the policies for them are shown in Figure 6.

- An "activity map" app can read the `location` column in `post`, but not any textual or image data. The post-to-image capability relationship is unavailable to it, so it cannot fetch images even for accessible posts.
- A "social collection" app gathers events from different social networks. It can read all posts and associated images from the "public" group.
- A "news" app has insert-only access to the service, which is sufficient for sharing news on Elgg. The policy fixes the `category` column of any inserted post to be "news", preventing it from posting into other categories.

Google Drive and client apps. The Google Drive proxy app in Earp provides a local service that mediates other apps' access to cloud storage, avoiding the need for OAuth. Client apps enjoy the benefits of cloud storage without having to worry about provider-specific APIs or managing access credentials. The proxy app presents a collection of file objects containing metadata (folder and file name) and data (file contents) to other apps. It services requests from client apps by making corresponding HTTPS requests to Google's remote service. We have ported two client apps to use the service.

- DriveNote is a note-taking app which stores notes on the user's Google Drive account via the local proxy. The proxy allows it to read/write files only in a dedicated

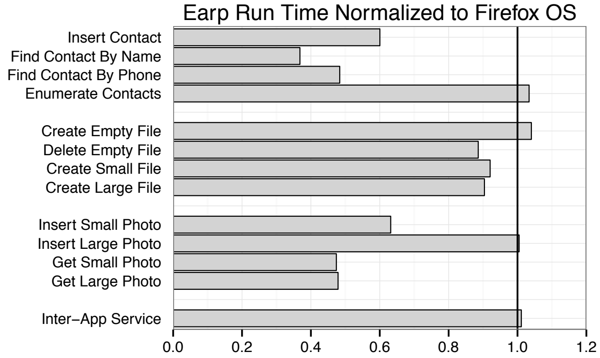


Figure 7: Microbenchmark results for storage and services. Smaller run time indicates better performance.

folder. Earp enforces this policy, ensuring that queries do not return files outside of this folder, and fixing the `folder` column on any update or insert operation.

- Gallery++ is a system utility, thus the Google Drive proxy app trusts it with access to all files. Gallery++ can scan and download all images stored on Google Drive.

7.2 Performance

We evaluate the performance of Earp on a Nexus 7 tablet, which has 2GB of DDR3L RAM and 1.5GHz quad-core Qualcomm Snapdragon S4 Pro CPU.

7.2.1 Microbenchmarks

We run various microbenchmarks to measure Earp's performance for storage and inter-app services. Figure 7 shows Earp's run time relative to Firefox OS.

DB-only workloads (contacts). We measure the time to insert new contacts, enumerate 500 contacts, and find a single contact matching a name or a phone number from the 500; the base line is the contacts manager in Firefox OS which uses IndexedDB. Earp outperforms the baseline for all workloads except enumerating contacts, where it is about only 3% slower.

Earp' performance is explained by its (1) directly using SQLite, while Firefox OS uses IndexedDB built on top of SQLite, (2) directly maping an object's fields into table columns, whereas IndexedDB uses expensive serialization to store the entire object, (3) using SQLite's built-in index support, whereas IndexedDB needs to create rows in an index table for all queryable fields of every object, (4) more complex data structure for contacts (six tables as opposed to a single serialized row for the baseline), which affords sophisticated access control but requires a bit more time to perform joins.

File-only workloads. We measure the time to create/delete empty files and write small (18KB)/large (3.4MB) files using Earp's directory API; the base line is Firefox OS' DeviceStorage API. Earp has comparable performance to the baseline, where the -11%∼4% difference in run time is due to different implementations of these APIs. Note that the measured times include event handling, e.g., dispatching to I/O threads and complete notification to the app.

	Baseline	Earp	slowdown
Elgg: read 50 posts	1623±102	1755± 99	8%
Elgg: upload 50 posts	5748±152	5888±117	2%
Google Drive: read 10 files	1310± 77	1392±120	6%
Google Drive: write 10 files	2828±217	2923±253	3%
Email: sync 200 emails	4725±433	4416±400	-6%

Table 1: Latency (msec) measured for macrobenchmarks on Earp applications.

DB-and-file workloads (photos). The measurements include inserting small, 18 KB, and large, 3.4 MB, photos with metadata, and retrieving them; the baseline is inserting/retrieving the same photo files and their metadata into the MediaDB library in Firefox OS, which uses IndexedDB. Earp largely outperforms the baseline, mostly because of the differences between SQLite and IndexedDB, as explained in the contacts experiments. When inserting large photos the run time is dominated by writing files so performance is very close (<1%) to the baseline.

Inter-app service. We measure the run time for retrieving 4,000 2 KB messages from a different app using Earp's inter-app service framework. The baseline uses Firefox OS' raw inter-app communication channel to implement an equivalent service, where requests are dispatched to Web worker threads (equivalent to Earp's thread pool). Figure 7 shows that Earp performs roughly the same as the baseline, and the time spent for access control (result filtering) is negligible.

7.2.2 Macrobenchmarks

Table 1 reports end-to-end latency for several real-world workloads described in Section 7.1.

Remote services. We measure the latency of client apps (Elgg client and DriveNote) accessing remote services (Elgg and Google Drive) by communicating with local proxy apps for these services. The baseline is the local proxy apps performing the same tasks by directly sending requests to their remote servers. The workloads include reading/uploading fifty posts with images via Elgg and reading/uploading ten 2KB text files via Google Drive. Table 1 shows that communicating with local proxy apps adds 3%∼8% latency, due to extra data serialization and event handling.

Email. We measure the latency of downloading 200 emails. The baseline is Firefox OS' email app which stores emails using IndexedDB. As shown in the "Email: sync" row of Table 1, Earp achieves similar performance storing the emails in an app-defined database.

8 Related work

Fine-grained, flexible protection on mobile platforms. TaintDroid [13] is a fine-grained taint-tracking system for Android. Several systems [21, 40, 45] rely on TaintDroid for fine-grained data protection. Pebbles [40] is most related to Earp: it modifies Android's SQLite and XML libraries and uses TaintDroid to discover app-level structured data across different types of storage. Pebbles relies on developers using certain design patterns consistently to infer the structure of data and it is implemented in an app-level library, not in the platform. Pebbles can help cooperative apps avoid mistakes, like preserving an attachment of a deleted email, but, unlike Earp, it cannot confine uncooperative apps.

Many systems extend Android to support more flexible and expressive permission policies [3, 4, 10, 11, 15, 30, 31, 52, 53] or mandatory access control [6, 39]. FlaskDroid [6] provides fine-grained data protection by implementing a design pattern that lets content providers present different views of shared data to different apps. FlaskDroid is limited to SQLite-based content providers and does not support cross-table capabilities. By contrast, Earp's framework supports all types of services, including proxies for remote servers. Moreover, in contrast to all existing systems, Earp integrates access-control policies with the data model itself, via capability relationships.

Fine-grained protection in databases. Traditional access control systems for relational databases [5, 7, 16, 20, 26, 32, 35] are based on users or roles with relatively static policies. Recently, IFDB [37] showed how decentralized information flow control (DIFC) can be integrated with a relational database. IFDB also discusses foreign key issues, but focuses on potential information leakage due to referential integrity enforcement. This is a very different problem than the one solved by Earp's capability relationships. The key contribution of Earp is identifying the relational model as the unified foundation for protecting data storage and sharing on mobile platforms.

Protection on Web platforms. BSTORE [8] provides a file system API to Web apps and uses tags to enable flexible access control on files. It is similar to Earp in that access control is enforced by a central reference monitor regardless of where the resource is hosted (local or remote). Unlike Earp, BSTORE's data abstraction is unstructured files.

Several systems enable flexible policies [25, 29], controlled object sharing [28, 33], or confinement [22, 23, 43] for JavaScript in a Web browser. Earp puts protection much lower in the system stack. For Web code interacting directly with the OS and other apps, Earp provides a unifying abstraction for both storage and inter-app services and adds access control directly into the data model.

Native relational stores. Like Earp, there are previous efforts to make relational data directly supported by the OS, notably Microsoft's cancelled project WinFS [50]. WinFS contained a database engine to natively support SQL, and implemented files and directories on top of the database. While WinFS had fine-grained access control, it was still based on per-object permissions.

WinFS was developed before mobile platforms become popular, and traditional desktop apps that rely on files suffered performance penalties due to database-managed metadata. Earp's database-centric approach fits the current mobile development practice where databases are the de facto storage hubs [40]. Crucially, Earp uses an unmodified file system (unlike WinFS) to store blob data and to provide compatibility file APIs that have no performance overhead.

9 Conclusion

Earp is a new mobile app platform built on a unified relational model for data storage and inter-app services. Earp directly exposes fine-grained, inter-related structured data as platform-level objects and mediates apps' access to these objects, enabling it to enforce app-defined access-control policies with simple building blocks, both old (ACLs) and new (capability relationships). Earp securely and efficiently supports key storage and sharing tasks of essential apps such as email, contacts manager, photo gallery, social networking and cloud storage clients, etc.

Acknowledgments. We thank the anonymous reviewers, Simon Peter, Michael Walfish, and our shepherd, Brad Karp, for valuable feedback and suggestions. This work was partially supported by NSF grants CNS-1223396 and 140894 and the NIH grant R01 LM011028-01.

References

[1] Android developers: URI permissions. `http://developer.android.com/guide/topics/security/permissions.html#uri`. [Online; accessed 21-September-2015].

[2] Android Developers: Using content providers. `http://developer.android.com/training/articles/security-tips.html#ContentProviders`. [Online; accessed 21-September-2015].

[3] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., AND VON STYP-REKOWSKY, P. AppGuard – real-time policy enforcement for third-party applications. Tech. Rep. A/02/2012, MPI-SWS, 2012.

[4] BERESFORD, A. R., RICE, A., SKEHIN, N., AND SOHAN, R. MockDroid: Trading privacy for application functionality on smartphones. In *International Workshop on Mobile Computing Systems and Applications (HotMobile)* (2011), ACM.

[5] BERTINO, E., JAJODIA, S., AND SAMARATI, P. Supporting multiple access control policies in database systems. In *IEEE Symposium on Security and Privacy* (1996).

[6] BUGIEL, S., HEUSER, S., AND SADEGHI, A.-R. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security Symposium* (2013).

[7] BYUN, J.-W., AND LI, N. Purpose based access control for privacy protection in relational database systems. *The VLDB Journal 17*, 4 (2008), 603–619.

[8] CHANDRA, R., GUPTA, P., AND ZELDOVICH, N. Separating web applications from user data storage with BSTORE. In *USENIX Conference on Web Application Development (WebApps)* (2010).

[9] CLOC – count lines of code. `http://cloc.sourceforge.net/`. [Online; accessed 17-September-2015].

[10] CONTI, M., NGUYEN, V. T. N., AND CRISPO, B. CRePE: Context-related policy enforcement for Android. In *Information Security Conference (ISC)* (2010).

[11] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium* (2011).

[12] Elgg - open source social networking engine. `https://www.elgg.org`. [Online; accessed 17-September-2015].

[13] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[14] How I exposed your private photos - Facebook private photo hack. `http://www.7xter.com/2015/03/how-i-exposed-your-private-photos.html`. [Online; accessed 17-September-2015].

[15] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium* (2011).

[16] FERRAIOLO, D. F., SANDHU, R., GAVRILA, S., KUHN, D. R., AND CHANDRAMOULI, R. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)4*, 3 (2001), 224–274.

[17] GEORGIEV, M., JANA, S., AND SHMATIKOV, V. Rethinking security of Web-based system applications. In *International World Wide Web Conference (WWW)* (2015).

[18] Google Drive API for Android: authorizing Android apps. `https://developers.google.com/drive/android/auth`. [Online; accessed 18-September-2015].

[19] Google Drive API for iOS: authorizing requests of iOS apps. `https://developers.google.com/drive/ios/auth`. [Online; accessed 18-September-2015].

[20] GUARNIERI, M., AND BASIN, D. Optimal security-aware query processing. In *International Conference on Very Large Data Bases (VLDB)* (2014).

[21] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *ACM Conference on Computer and Communications Security (CCS)* (2011).

[22] HOWELL, J., PARNO, B., AND DOUCEUR, J. R. Embassies: Radically refactoring the Web. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[23] INGRAM, L., AND WALFISH, M. Treehouse: JavaScript sandboxes to help web developers help themselves. In *USENIX Annual Technical Conference* (2012).

[24] iOS developer library: App extension programming guide. `https://developer.apple.com/library/ios/documentation/General/Conceptual/ExtensibilityPG/index.html#//apple_ref/doc/uid/TP40014214`. [Online; accessed 17-September-2015].

[25] Jayaraman, K., Du, W., Rajagopalan, B., and Chapin, S. J. Escudo: A fine-grained protection model for web browsers. In *IEEE International Conference on Distributed Computing Systems (ICDCS)* (2010).

[26] Jeloka, S., et al. *Oracle Label Security Administrator's Guide*, release 2 (11.2) ed. Oracle Corporation, 2009.

[27] Li, T., Zhou, X., Xing, L., Lee, Y., Naveed, M., Wang, X., and Han, X. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[28] Meyerovich, L. A., Felt, A. P., and Miller, M. S. Object views: Fine-grained sharing in browsers. In *International World Wide Web Conference (WWW)* (2010).

[29] Meyerovich, L. A., and Livshits, B. Conscript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy* (2010).

[30] Nauman, M., Khan, S., and Zhang, X. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)* (2010).

[31] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically rich application-centric security in Android. *Security and Communication Networks 5*, 6 (2012), 658–673.

[32] Osborn, S., Sandhu, R., and Munawer, Q. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security (TISSEC)3*, 2 (2000), 85–106.

[33] Patil, K., Dong, X., Li, X., Liang, Z., and Jiang, X. Towards fine-grained access control in JavaScript contexts. In *IEEE International Conference on Distributed Computing Systems (ICDCS)* (2011).

[34] Ritchie, D. M., and Thompson, K. The UNIX time-sharing system. *Communications of the ACM (CACM) 17*, 7 (1974).

[35] Rizvi, S., Mendelzon, A., Sudarshan, S., and Roy, P. Extending query rewriting techniques for fine-grained access control. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2004).

[36] Saltzer, J. H. Protection and the control of information sharing in multics. *Communications of the ACM (CACM) 17*, 7 (1974).

[37] Schultz, D., and Liskov, B. IFDB: Decentralized information flow control for databases. In *ACM European Conference in Computer Systems (EuroSys)* (2013).

[38] Shahriar, H., and Haddad, H. Content provider leakage vulnerability detection in Android applications. In *SIN* (2014).

[39] Smalley, S., and Craig, R. Security enhanced (SE) Android: Bringing flexible MAC to Android. In *Network and Distributed System Security Symposium (NDSS)* (2013).

[40] Spahn, R., Bell, J., Lee, M. Z., Bhamidipati, S., Geambasu, R., and Kaiser, G. Pebbles: Fine-grained data management abstractions for modern operating systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[41] Sqlite foreign key support. `https://www.sqlite.org/foreignkeys.html`. [Online; accessed 18-September-2015].

[42] The virtual table mechanism of SQLite. `https://www.sqlite.org/vtab.html`. [Online; accessed 17-September-2015].

[43] Stefan, D., Yang, E. Z., Marchenko, P., Russo, A., Herman, D., Karp, B., and Mazieres, D. Protecting users by confining JavaScript with COWL. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[44] Sun, S.-T., and Beznosov, K. The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[45] Tang, Y., Ames, P., Bhamidipati, S., Bijlani, A., Geambasu, R., and Sarda, N. CleanOS: Limiting mobile data exposure with idle eviction. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2012).

[46] HTML5 trumping iOS among app developers in emerging mobile markets. `http://www.zdnet.com/article/html5-trumping-ios-among-app-developers-in-emerging-mobile-markets/`. [Online; accessed 17-September-2015].

[47] Survey: Most developers now prefer HTML5 for cross-platform development. `http://techcrunch.com/2013/02/26/survey-most-developers-now-prefer-html5-for-cross-platform-development/`. [Online; accessed 17-September-2015].

[48] VIENNOT, N., GARCIA, E., AND NIEH, J. A measurement study of Google Play. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2014).

[49] WANG, R., XING, L., WANG, X., AND CHEN, S. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[50] Introducing "Longhorn" for developers, Chapter 4: Storage. `https://msdn.microsoft.com/en-us/library/Aa479870.aspx`. [Online; accessed 17-September-2015].

[51] XING, L., BAI, X., LI, T., WANG, X., CHEN, K., LIAO, X., HU, S., AND HAN, X. Cracking app isolation on Apple: Unauthorized cross-app resource access on MAC OS X and iOS. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[52] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security Symposium* (2012).

[53] XU, Y., AND WITCHEL, E. Maxoid: Transparently confining mobile applications with custom views of state. In *ACM European Conference in Computer Systems (EuroSys)* (2015).

[54] ZHOU, Y., AND JIANG, X. Detecting passive content leaks and pollution in Android applications. In *Network and Distributed System Security Symposium (NDSS)* (2013).

iCellular: Device-Customized Cellular Network Access on Commodity Smartphones

Yuanjie Li[1], Haotian Deng[2], Chunyi Peng[2], Zengwen Yuan[1], Guan-Hua Tu[1], Jiayao Li[1], Songwu Lu[1]

[1] University of California, Los Angeles [2] The Ohio State University

Abstract

Exploiting multi-carrier access offers a promising direction to boost access quality in mobile networks. However, our experiments show that, the current practice does not achieve the full potential of this approach because it has not utilized fine-grained, cellular-specific domain knowledge. In this work, we propose *iCellular*, which exploits low-level cellular information at the device to improve multi-carrier access. Specifically, *iCellular* is proactive and adaptive in its multi-carrier selection by leveraging existing end-device mechanisms and standards-complaint procedures. It performs adaptive monitoring to ensure responsive selection and minimal service disruption, and enhances carrier selection with online learning and runtime decision fault prevention. It is readily deployable on smartphones without infrastructure/hardware modifications. We implement *iCellular* on commodity phones and harness the efforts of *Project Fi* to assess multi-carrier access over two US carriers: T-Mobile and Sprint. Our evaluation shows that, *iCellular* boosts the devices with up to 3.74x throughput improvement, 6.9x suspension reduction, and 1.9x latency decrement over the state-of-the-art selection scheme, with moderate CPU, memory and energy overheads.

1 Introduction

Mobile Internet access has become an essential part of our daily life with our smartphones. From the user's perspective, (s)he demands for high-quality, anytime, and anywhere network access. From the infrastructure's standpoint, carriers are migrating towards faster technologies (*e.g.*, from 3G to 4G LTE), while boosting network capacity through dense deployment and efficient spectrum utilization. Despite such continuous efforts, no single carrier can ensure complete coverage or highest access quality at any place and anytime.

In addition to infrastructure upgrades from carriers, a promising alternative is to leverage multiple carrier networks at the end device. In reality, most regions are covered by several carriers (say, Verizon, T-Mobile, Sprint, and AT&T in the US). With multi-carrier access, the device may select the best carrier over time and improve its overall access quality. The exciting Google *Project Fi* [26] has taken the lead to provide 3G/4G multi-carrier access in practice. Other similar efforts through universal SIM card include Apple SIM [14] and Samsung e-SIM [24]. The upcoming 5G standards also seek to support multiple, heterogenous access technologies [34].

Our empirical study shows that, the full benefits of multi-carrier access can be constrained by today's design. We examine Google *Project Fi* over two carriers (T-Mobile and Sprint), and discover three issues, all of which are independent of its excellent implementations (§3): (P1) The anticipated switch is never triggered even when the serving carrier's coverage is pretty weak; (P2) The switch takes rather long time (tens of seconds or minutes) and prolongs service unavailability; and (P3) the device fails to choose the high-quality network (*e.g.*, selecting 3G with weaker coverage rather than 4G with stronger coverage).

It turns out that, the above issues can be effectively addressed by using low-level cellular information (*e.g.*, available carriers, which carriers to scan, and radio/QoS profile for each carrier) and mechanisms. However, such fine-grained knowledge is not available to commodity phones in their default operations. This is rooted in the fundamental design of 3G/4G networks. With the single-carrier scenario in mind, 3G/4G follows the design paradigm of *"smart core, dumb end"*. It thus does not expose its low-level information to the device in normal operations. For multi-carrier access, however, end intelligence is a necessity, since individual carrier does not have global view on all carriers, which can only be constructed at the device through accessing low-level cellular events. Without using such knowledge, today's carrier selection could encounter issues P1-P3.

While the problem can be solved by the future architecture redesign (say, 5G), it usually takes years to accomplish. Instead, we seek to devise a solution that works with the current 3G/4G network, in line with the ongoing industrial efforts, *e.g.*, Google *Project Fi*, Apple SIM and Samsung e-SIM. Specifically, we address the following problem: *Can we leverage low-level cellular*

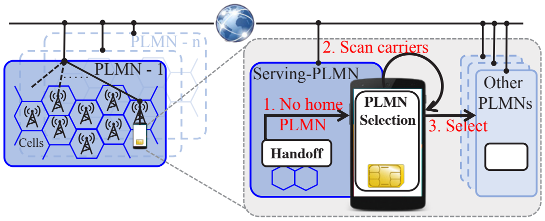


Figure 1: Multi-carrier network access (left) and inter-carrier switch via PLMN selection (right).

information and mechanisms at the device to further improve multi-carrier access? Our study yields a positive answer.

We propose *iCellular*, a client-side service to let mobile devices customize their own cellular network access. Complementing the design of *Project Fi*, *iCellular* further leverages low-level, runtime cellular information at the device during its carrier selection. *iCellular* is built on top of current 3G/4G mechanisms at the device, but applies cross-layer adaptations to ensure responsive multi-carrier access with minimal disruption. To facilitate the device to make proper decisions, *iCellular* exploits online learning to predict the performance of heterogenous carriers, and provides built-in strategies for better usability. It further safeguards access decisions with fault prevention techniques. We implement *iCellular* on commodity phone models (Nexus 6 and Nexus 6P) and assess its performance with *Project Fi*. Our evaluation shows that, *iCellular* can achieve 3.74x throughput improvement and 1.9x latency reduction on average by selecting the best mobile carrier. Meanwhile, *iCellular* has negligible impacts on the device's data service and OS resource utilization (less than 2% CPU usage), approximates the lower bounds of responsiveness and switch disruption, and shields its selection strategies from decision faults.

The rest of the paper is organized as follows. §2 introduces the background. §3 describes our findings and uncovers root causes of multi-carrier access. §4, §5, and §6 present the design, implementation and evaluation of *iCellular*, respectively. §7 discusses remaining issues, and §8 presents the related work. §9 concludes the work.

2 Mobile Network Access Primer

A cellular carrier deploys and operates its mobile network (called public land mobile network or PLMN) to offer services to its subscribers. Each PLMN has many cells across geographical areas. Each location is covered by multiple cells within one PLMN and across several PLMNs (*e.g.*, Verizon, AT&T, T-Mobile, Sprint).

Single-carrier network access. Today's cellular network is designed under the premise of single-carrier access. A mobile device is supposed to gain access directly from its home PLMN. It obtains radio access from the serving cell and further connects to the core carrier network and the external Internet, as shown in the left plot of Figure 1. When the current cell can no longer serve the device (*e.g.*, out of its coverage), the device is migrated to another available cell within the same PLMN. This is called handoff.

Roaming between carriers. When the home PLMN cannot serve its subscribers (*e.g.*, in a foreign country), the device may roam to other carriers (visiting networks). This is realized through the PLMN selection procedure between carriers [12], which is a mandatory function for all commodity phones. It supports both automatic (based on a pre-defined PLMN priority list) and manual modes. As shown in the right plot of Figure 1, once triggered by certain events (*e.g.* no home PLMN service), PLMN selection should first scan the available carriers, and then choose one based on the pre-defined criteria (*e.g.* preference) or the user manual operation. If the device decides to switch, it will deregister from the current carrier network and then register to a new one. In this process, network access may be temporarily unavailable. This is acceptable since inter-carrier switch is assumed to be infrequent, thus having limited impacts.

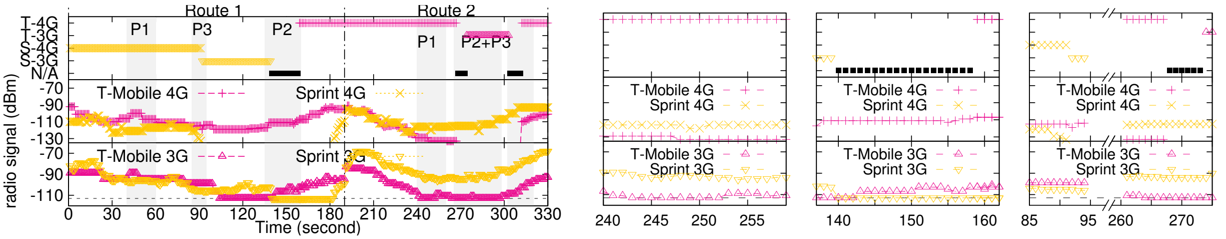
Multi-carrier access with universal SIM card. Recent industrial efforts aim at providing mobile device access to multiple carriers with a single SIM card. They include Google Project Fi [26], Apple SIM [14], and Samsung e-SIM [24]. With the SIM card, the device can access multiple cellular carriers (*e.g.*, T-Mobile and Sprint in *Project Fi*). Given only one cellular interface, the device uses one carrier at a time.

3 Multi-carrier Access: Promises & Issues

We run experiments to quantify the benefits of multi-carrier access, and identify the downsides of the today's efforts. The identified limitations are independent of implementations, but rooted in the 3G/4G design.

Methodology. We conduct both controlled experiments and a one-month user study using two Nexus 6 phones with Google *Project Fi* [26], which was released in May 2015. *Project Fi* provides access to two U.S. carriers (T-Mobile and Sprint) at this time. It develops an automatic carrier selection on commodity phones using a proprietary mechanism. Unfortunately, details of its switching algorithm have not been published. We contacted *Project Fi* team and learned that this algorithm aims at optimizing consumer experience, and considers network performance, battery usage and data activity during selection. We further inferred its decision and execution strategies from our experiments.

In each controlled test, we use a Nexus 6 phone with a *Project Fi* SIM card, and test with *Project Fi*'s automatic carrier selection mode. We walk along two routes



(a) An example log over two walking routes (b) P1: no switch (c) P2: disruption (d) P3: unwise switch

Figure 2: An example log for serving carriers and networks and three problematic instances through *Project Fi*.

within the campus buildings at UCLA and OSU at the idle mode (no data/voice, screen off). We walk slowly (< 1 m/s) and record the serving carrier ("T" for T-Mobile, "S" for Sprint) and its network type (4G or 3G) per second. Meanwhile, we carry other accompanying phones to record the radio signal strength of each access option (T-4G, T-3G, S-4G, S-3G). We run each test 10 times and similar results are consistently observed in all the tests. In the user study (07/31/15 to 09/02/15), we use the *Project Fi*-enabled phone as usual and collect background device and cellular events with *MobileInsight*, an in-phone cellular monitoring tool [4]. We have collected 4.9GB logs with *MobileInsight* in total, with 274,351 messages from radio resource control (RRC), 16,470 messages from mobility management (MM), and 5,365 messages from session management (SM). We next present the results from the controlled experiments as motivating examples. The user study to be described in §4 and §6 confirms that these issues are common in practice.

3.1 Motivating Examples

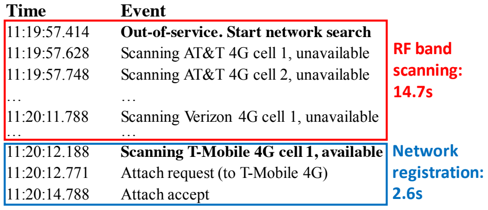
Merits of multi-carrier access. We first verify that exploiting multiple carriers is indeed beneficial to service availability and access quality. Figure 2a shows the results from the controlled experiments over two routes. On the first route [0s,190s), Sprint gradually becomes weaker and then fades away, but its dead zone is covered by T-Mobile; On the second route [190s, 330s], in contrast, Sprint offers stronger coverage, even at locations with extremely weak coverages from T-Mobile. Multi-carrier access indeed helps to enhance network service availability by boosting radio coverage. For example, in [160s, 180s], the phone switches to T-Mobile and retains its radio access while Sprint is not available. Moreover, we confirm that it further improves data access throughput and user experiences. The *Project Fi* indeed offers a major step forward on mobile Internet access.

Our examples further reveal three issues, which demonstrate that the benefits of multi-carrier access have not been fully achieved.

P1. No anticipated inter-carrier switch. It is desirable for the device to migrate to another available carrier network for better access quality, when the device perceives degraded quality from its current, serving carrier. However, our experiments show that, the device often gets stuck in one carrier network, and misses the better network access (e.g., during [40s, 60s] and [240s, 260s] of Figure 2). As shown in Figure 2b, T-Mobile experiences extremely weak radio coverage (< -130 dBm in 4G and < -110 dBm in 3G), but the phone never makes any attempt to move to Sprint, regardless of how strong Sprint's radio signal is. As a result, the device fails to improve its access quality. Moreover, we find that the expected switch often occurs until its access to the original carrier (here, T-Mobile) is lost. This is rooted in the fact that the inter-carrier switch is triggered when the serving carrier fails. Therefore, the device becomes out of service in this scenario, although better carrier access remains available.

P2. Long switch time and service disruption. Even when inter-carrier switch is eventually triggered, it may disrupt access for tens of seconds or even several minutes (see Figure 6 for the user-study results). In the example of Figure 2c, the phone starts Sprint→T-Mobile roaming at the 140th second, but it takes 17.3s to gain access to T-Mobile 4G. This duration is much longer than the typical handoff latency (possibly several seconds [42]). It is likely to halt or even abort any ongoing data service. We look into the event logs (Figure 3) to examine why the switch is slow. It turns out that, most of the switch time is wasted on an *exhaustive* scanning of all possible cells, including nearby cells from AT&T and Verizon. In this example, it spends 14.7s on radio-band scanning and 2.6s on completing the registration (attachment) to the new carrier (here, T-Mobile). Note that, such heavy scanning overhead is not incurred by any implementation glitch. Instead, it is rooted in the *Project Fi*'s design, which selects a new carrier network only after an exhaustive scanning process. In this work, we want to show that such large latency is unnecessary. It can be reduced without compromising inter-carrier selection.

P3. Unwise decision and unnecessary performance degradation. Our next finding is that, the device fails



```
Time            Event
11:19:57.414    Out-of-service. Start network search
11:19:57.628    Scanning AT&T 4G cell 1, unavailable
11:19:57.748    Scanning AT&T 4G cell 2, unavailable
…               …
11:20:11.788    Scanning Verizon 4G cell 1, unavailable
…               …
11:20:12.188    Scanning T-Mobile 4G cell 1, available
11:20:12.771    Attach request (to T-Mobile 4G)
11:20:14.788    Attach accept
```

RF band scanning: 14.7s
Network registration: 2.6s

Figure 3: Event logs during P2 (disruption) of Fig. 2c.

to migrate to the better choice, thus unable to enjoy the full benefits of multi-carrier access. The phone often moves to 3G offered by the same carrier, rather than the 4G network from the other carrier that yields higher speed. Figure 2d illustrates two such instances. After entering an area without Sprint 4G at the 91st second, the device switches to Sprint 3G, despite stronger radio signals from T-Mobile 4G. This indicates that the intra-carrier handoff is preferred over the inter-carrier switch in practice. Unfortunately, such a preference choice prevents the inter-carrier switch from taking effect. Even worse, obstacles still remain even when the network access to the original carrier has been shortly disrupted. For instance, during [267s, 273s], the original carrier (T-Mobile 3G) is still chosen. In this case, T-Mobile 4G and 3G networks almost have no coverage. In short, the device acts as a single-carrier phone in most cases, even with the multi-carrier access capability. Inter-carrier switch is not triggered as expected.

3.2 Insights

The above examples also shed lights on how to solve the three problems. The key is to leverage low-level cellular information and mechanisms at the device when selecting access from multiple carriers.

Specifically, performing the anticipated switch (P1) states that, the device performs inter-carrier switch upon detecting a better carrier, even when the serving carrier is still available. This further requires the device to learn all available carriers and their quality at runtime. Note that such information can be obtained from the low-level cellular events. However, the default operation on commodity phones will not do so. Moreover, the naive approach of forcing the phone to proactively scan other carriers at any time may lead to temporary disconnection from the current carrier network. We elaborate on how we address these issues in §4.1.

To reduce the switch time (P2), the device should refrain from exhaustive search of all carriers at all times. This requires the device to perform fine-grained control on which carriers should be scanned. It can be done by configuring the low-level mechanism for monitoring.

To make a wise selection decision (P3), the device should treat all intra-carrier handoffs and inter-carrier switches equally, and select the best carrier network. This requires the device to directly initiate the inter-carrier switch when needed. This also calls for leveraging the low-level cellular mechanism.

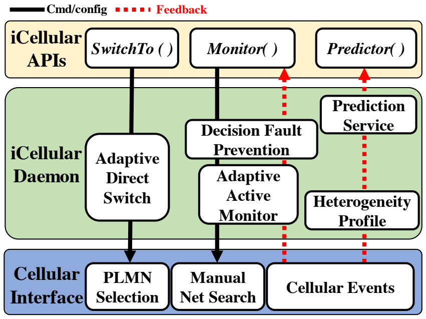


Figure 4: *iCellular* system architecture.

In summary, low-level domain knowledge can be exploited to effectively address all three issues. However, the default operation mode on commodity phones does not expose such fine-grained cellular information and mechanisms to higher layers. The reason is that, the 3G/4G network follows the design paradigm of "smart core, dumb end" with the single-carrier usage scenario in mind. The end device does not need to exploit such information when selecting its carrier access. Since such low-level, cellular-specific domain knowledge is not available for the default operation mode, it might be the reason why *Project Fi* has not explored this direction in its current design.

4 *iCellular* Design

We now present *iCellular*, which explores an alternative dimension to improve multi-carrier access. *iCellular* complements the design of *Project Fi* by leveraging low-level cellular information and mechanisms. It seeks to further empower the end device to have more control on its carrier selection, while addressing the issues in §3.1.

For incremental deployability, *iCellular* is built on top of the PLMN selection [11, 12], a standardized mechanism mandatory on all phones. Note that, however, the basic PLMN selection suffers from similar issues in §3.1: migrating to other carriers is not preferred unless the home carrier fails (P1); the exhaustive scanning (P2) and the preferable intra-carrier handoffs (P3) are still in use. The reason is that, the default PLMN selection scheme is designed under the premise of single-carrier access. While roaming to other carriers is allowed, it is not preferred by the home carrier unless it fails to offer network access to its subscribers. So the basic PLMN selection has the following features: (1) *Passive triggering/monitoring:* When being served by one carrier, the device should not monitor other carriers or trigger the selection until the current one fails (*i.e.*, out of coverage); (2) *Network-controlled selection:* The device should select the new carrier based on the preferences pre-defined by the home carrier and stored in the SIM card; (3) *Hard switch:* The device should deregister from the old car-

Function	Method	Cellular Events	Type
Active monitor (§4.1)	Disruption avoidance	Paging	Meas
		Paging cycle	Config
	Minimal search	Radio meas	Meas
		RRC SIB 1	Config
Prediction service (§4.3)	QoS profile	EPS/PDP setup	Config
	Radio profile	RRC reconfig	Config
Decision fault prevention (§4.4)	Access control	RRC SIB1	Config
	Interplay with net mobility	Cell reselection in RRC SIB 3-8	Config
	Function completeness	GMM/EMM location update	Config

Table 1: Cellular events used in *iCellular*.

rier first, and then register to the new one. We thus need to adapt the PLMN selection scheme to the multi-carrier context by using low-level cellular events.

Figure 4 illustrates an overview of *iCellular*. In brief, *iCellular* systematically enhances the devices' role in every step of inter-carrier switch with runtime cellular information, spanning triggering/monitoring, decision making and switch execution. To be incrementally deployable on commodity phones, we build *iCellular* on top of the existing mechanisms from the phone's cellular interface [7]. We exploit the freedom given by the standards, which allow devices to tune configurations and operations to some extent. To ensure responsiveness and minimal disruption, *iCellular* applies cross-layer adaptations over existing mechanisms (§4.1 and §4.2). To facilitate the devices to make wise decisions, *iCellular* offers cross-layer online learning service to predict network performance (§4.3), and protects devices from decision faults (§4.4). To enable adaptation, prediction and decision fault prevention, *iCellular* incorporates realtime feedbacks extracted from low-level cellular events. Different from approaches using additional diagnosis engine (*e.g.*, QXDM [37]) or software-defined radio (*e.g.*, LT-Eye [30]), we devise an in-phone mechanism to collect realtime cellular events (§4.5, cellular events are summarized in Table 1). These components are designed to be scalable, without incurring heavy signaling overhead to both the device and the network.

4.1 Adaptive Monitoring

To enable device-initiated selection, the first task is to gather runtime information on available carrier networks. This is done through *active monitoring*. It allows a device to scan other carriers even while being served by one. This would prevent the device from missing a better carrier network (P1 and P3 in §3). For this purpose, the only viable mechanism on commodity phones is the `manual network search` [12]. It was designed to let a device manually scan all available carriers. Once initiated, the device scans neighbor carriers' frequency bands, extracts the network status from the broadcasted system information block, and measures their radio quality. No extra signaling overhead is incurred, since the active monitoring approach does not activate signaling exchanges between the device and the network. To be incrementally deployable, we decide to realize active monitoring on top of the manual network search.

Note that naive manual search does not satisfy properties of minimal-disruption and responsiveness. First, scanning neighbor carriers may disrupt the network service. The device has to re-synchronize to other carriers' frequency bands, during which it cannot exchange traffic with the current carrier. Second, it is *exhaustive* to all carriers by design. Even if the device is not interested in certain carriers (*e.g.*, no roaming contract), this function would still scan them, thus delaying the device's decision and wasting more power. The challenge is that, both issues cannot be directly addressed with application-level information only. *iCellular* thus devises cross-layer adaptions for both issues.

Disruption avoidance. To minimize disruptions on ongoing services, *iCellular* schedules scanning events only when the device has no application traffic delivery. This requires *iCellular* to monitor the uplink and downlink traffic actvities. While the uplink one can be directly known from the device itself, the status for downlink traffic is hard to predict. Traffic may arrive while the device has re-synchronized to other carriers' cells. If so, its reception could be delayed or even lost.

iCellular prevents this by using the low-level cellular event feedback. We observe that in the 3G/4G network, the downlink data reception is regulated by the periodical paging cycle (*e.g.*, discontinuous reception in 4G [9,38]). To save power, the 3G/4G base station assigns inactivity timers for the device. The device periodically wakes up from the sleep mode, monitors the paging channel to check downlink data availability, and moves to the sleep mode again if no traffic is coming. *iCellular* obtains this cycle configuration from the radio resource control (RRC) messages, and schedules its scanning operations only during the sleep mode. Figure 5 shows our one-month logs of 4G per-cell search time at a mobile device with *Project Fi*. It shows that, 79.2% of cells can be scanned in less than one paging cycle. Others need more cycles to complete the scanning. With this design, no paging event is interrupted by monitoring.

One valid concern is that, the monitoring results may become obsolete due to continuous data transmissions, thus leading to wrong decisions. This is unlikely to happen in practice for two reasons. First, most traffic tends to be bursty, which leaves sufficient idle period for background monitoring. Second, network performance tends to vary smoothly, and stale monitoring results do not affect the final selection decision. Furthermore, *iCellular* compares the elapsed time between the decision making and the measurement. Obsolete measurements outside the time window (say, 1 minute) will not be used.

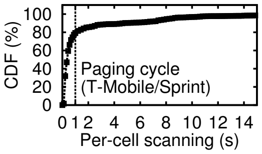


Figure 5: Cell scan time.

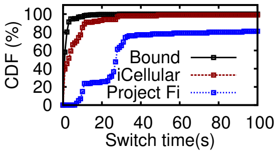


Figure 6: Switch time.

Minimal search. Instead of exhausting all carrier networks, *iCellular* scales the monitoring by restricting the manual search only to those specified by the device. To realize this idea, the practical issue is that no such option is available in the manual network search mechanism. We thus leverage adaptation of the PLMN preference. Given the list of carrier networks of interests , *iCellular* configures the cellular interface to let the manual network search scan these carriers first. This is achieved by assigning them with highest PLMN preferences. During the manual search, *iCellular* listens to the cellular events to see which carrier is being scanned. These events include the per-cell radio quality measurements, and its system information block with PLMN identifiers. Once *iCellular* detects that the device has finished scanning of the device-specified carriers, it terminates the manual network search function.

Monitoring-decision parallelism. Sometimes there is no need to complete all the monitoring to determine the target carrier network. For example, if the user prefers 4G, it can decide to switch whenever a good 4G is reported, without waiting for 3G results. To support this, *iCellular* allows devices to make decisions with partial results, thus further accelerating the process. Instead of waiting for all scanning results, *iCellular* triggers the decision callback whenever new results are available.

4.2 Direct Inter-carrier Switch

iCellular aims at reducing the disruption time incurred by inter-carrier switching as much as it can. We find that, there is enough room for this because *most service disruption time is caused by frequency band scanning* (§3). With the active monitoring function, *iCellular* does not need to scan the carrier networks during switch. Specifically, given a target carrier network, *iCellular* makes a direct switch by configuring the target carrier with highest PLMN preference. It then triggers a manual PLMN selection to the target carrier network. This way, the device would directly switch to the target without unnecessary scanning.

We next show how *iCellular* approximates to the lower bound of the switch time. In cellular networks, switching to another network requires at least deregistration from the old network (detach), and registration to the new network (attach). According to [10], the detach time is negligible, since the device can detach directly without interactions to the old carrier network. So the minimal disruption time in switch is roughly equal to the attach time, *i.e.*, $T_{switch,min} \approx T_{attach}$. For *iCellular*, no extra attempts to other carrier networks are made. Since it is on top of the PLMN selection, the scanning of the target carrier still remains. Therefore, the switch time is

$$T_{switch,iCellular} = n_t T_t + T_{attach} = n_t T_t + T_{switch,min} \quad (1)$$

where n_t and T_t are the cell count and per-cell scanning time for the target carrier network, respectively. Compared with the attach time, this extra overhead is usually negligible in practice. Figure 6 verifies this with our one-month background monitoring results in *Project Fi*. It shows that, *iCellular* indeed approximates the lower bound, despite this minor overhead.

4.3 Prediction for Heterogeneous Carriers

To decide which carrier network to switch to, the device may gather performance information on each carrier network. Ideally, the device needs to measure every available carrier network's current performance (*e.g.*, latency or throughput) and make decisions. Unfortunately, this is deemed impossible. The device can only measure the serving network's performance; other candidates' performances cannot be measured without registration.

Given this fact, *iCellular* decides to assist the device to predict each carrier's performance. Our predication is based on the regression tree algorithm [19]. It models the network/application performance (y) as a function of a feature vector (x_1,x_2), where x_1 is runtime radio measurement and x_2 is carrier network profiles (elaborated below). The model is established using a pre-stored tree at bootstrap and then recursively updated with new online samples. Note that radio measurement alone is insufficient to predict performance, because different carriers may apply heterogeneous radio technologies and resource configurations. Our prediction works as follows.

Prediction metric (y). This metric is used to rank the performances of all available networks. We explore both network-level (link throughput, radio latency) and application-level ones (*e.g.*, web loading latency, video suspension time). They are obtained from both network and application events, for example, Appendix B shows how to obtain app-specific metrics. We want to point out that the app-specific metric often leads to the same selection decision (see the evaluation §6). This is because the performance characteristics of a carrier network tend to have consistent impacts on all applications.

Training sample collection. The training sample (x,y) for a network is collected in the background, without interrupting the device's normal usage. A new training sample is collected when a new observation of the performance metric y is generated (*e.g.*, throughput from physical layer, loading time for Web-page download, latency per second for VoIP). In the meantime, radio mea-

Profile		Sprint		T-Mobile	
		Value	Prob	Value	Prob
QoS	Traffic class	Background	100%	Interactive	97.5%
	Delay class	4 (best effort)	100%	1	100%
	Max dlink rate	200Mbps	100%	256Mbps	100%
	Max ulink rate	200Mbps	100%	44Mbps	100%
Radio	Duplex type	TDD	88.3%	FDD	100%
	Paging cycle	100–200ms	81.5%	100ms	99.4%
	Handoff priority	2/3/6	100%	2/3/6	100%

Table 2: Heterogeneous cellular network profiles.

surement and network profiles for the serving network are recorded as $x = (x_1, x_2)$. For the radio quality x_1, *iCellular* extracts the serving network's RSRP (if 4G) or RSCP (if 3G) from the runtime active monitor (§4.1). For the network profile, *iCellular* currently collects two types (Table 2): (1) QoS profile from the data bearer context in session management, which includes the delay class and peak/maximum throughput; (2) radio parameters from the RRC configuration message, which includes the physical and MAC layer configurations. Note that the device cannot gain these profiles at runtime without registration to the carrier network of interest. To address this issue, we observe that network profiles are quite predictable. This is validated by our 1-month user study. Table 2 lists the predictability of some parameters from this log. For each parameter, we choose the one with the highest probability, and shows its occurrence probability. Note that, most QoS and radio configurations are invariant of time and location. The reason is that, the carriers tend to apply well-tested operation rules (*e.g.*, link adaptation and scheduling), with minor tunings to each base station/controller. As a result, we only store a set of unique values, and reuse it for all the applicable samples until changes are found.

Online predication and training. *iCellular* uses an online regression tree algorithm [19] as its predictor. The predictor is represented as a tree, with each interior node as a test condition over x (radio measurements and profile fields). Each decision is made upon the arrival of the feature vector x. It estimates the per-network metric y and selects the one with the highest rank.

iCellular updates the predictor's decision tree in the online fashion when a new sample arrives. At the bootstrap phase, it pre-stores a regression tree based on an offline training as the basis. Given a new sample (x, y), *iCellular* first determines whether a predictor update is needed. It runs the existing predictor over the heterogeneity information and runtime radio measurements, and obtains an estimated metric $y\prime$. If $|y - y\prime| = \min_{z \in leaf} |y - z|$, which implies the current sample fits well with the existing model, no update is needed. Otherwise, the predictor is updated as follows. Given the new sample and the existing tree, *iCellular* searches a new field (measurement or profile) that best splits the samples by minimizing the impurities in the two children nodes (based on the least-square criterion). Given this new split, we create a new pair of leaves for this new field, and completes the update of the prediction tree. Note that *iCellular* responds to new changes, and does not need to permanently store all training samples. This way, *iCellular* is scalable in storage and computation.

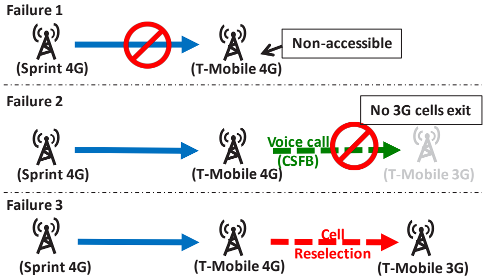


Figure 7: Three types of improper switch decisions.

4.4 Decision Fault Prevention

Letting a device customize its access strategy can be a double-edged sword. With improper strategies, the device may make faulty switch decisions and cause unexpected service disruption. Figure 7 shows three categories of failures caused by decision faults, all of which can only be detected with low-level cellular information:

Failure 1: No network access. Certain networks may be temporarily inaccessible. For example, our user study reports that, a Sprint 4G base station experiences a 10-min maintenance, during which access is denied.

Failure 2: No voice service. In some scenarios, the target carrier network cannot provide complete voice services. Figure 8 shows an instance from our user study. T-Mobile provides its voice service using circuit-switched-fall-back (CSFB), which moves the device to 3G for the voice call. However, there exist areas not covered by T-Mobile 3G (*e.g.*, signal strength lower than -95dBm according to [8]). In this scenario, the user in Sprint 4G should not switch to T-Mobile 4G, which cannot support voice calls without the 3G infrastructure.

Failure 3: Unexpected low-speed data service. The user selection may not be honored by the individual carrier's handoff rules. Figure 9 reports an instance from our user study. The user under Sprint 4G may decide to switch to one T-Mobile 4G. However, under the same condition, T-Mobile's mobility rules (*e.g.*, cell reselection [11]) would switch its 4G users to its 3G. In this case, the user's decision to T-Mobile 4G is improper, because the target network (T-Mobile 3G) is not preferred, and this switch incurs unnecessary disruptions.

To prevent decision faults, *iCellular* chooses to safeguard the device's decisions from those faulty ones. It checks whether each carrier network has any of the above problems, and excludes such carriers from the monitoring results. This prevents the device from switch-

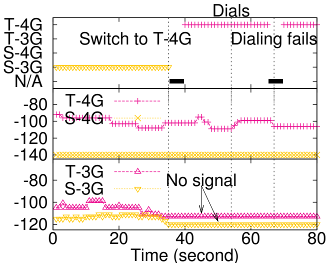


Figure 8: Switch to a network with no voice support.

Time	Event
17:49:07.520	**Deregister from Sprint 4G**
17:49:13.433	Scanning T-Mobile 4G cell 1, available
17:49:13.508	**Cell reselection config in SIB6: switch to 3G when RSRP$_{4G}$<-120dBm**
17:49:15.142	Attach accept
17:49:20.106	**RSRP$_{4G}$= -122dBm**
17:49:21.326	**Cell reselection to T-Mobile 3G**

Figure 9: Interplay between user and network's mobility.

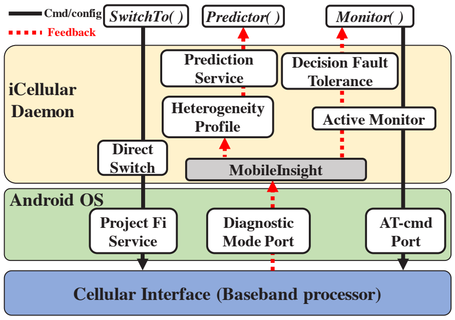


Figure 10: Overview of *iCellular* implementation.

ing to these carrier networks. To this end, *iCellular* first profiles each carrier's low-level access-control list from the RRC system-info-block message [9], data/voice preference configuration from registration/location update messages [10], and the network-side mobility rules from the RRC configuration message [9,11]. At runtime, for each candidate carrier, it checks if it is in the forbidden list (Failure 1), has no voice service with satisfactory 3G radio quality (Failure 2), or has satisfied mobility rules for further switch (Failure 3). If any condition is satisfied, it would be removed from the monitoring list.

4.5 Cellular Events Collection

As shown in §4.1-§4.4, *iCellular* relies on low-level cellular events to perform cross-layer adaptations over the existing mechanisms, predict the network performance, and avoid possible switch faults. The cellular events include the signaling messages exchanged between the device and the network, and radio quality/load measurements. Table 1 summarizes the events required by *iCellular*. Note that some events (*e.g.*, paging) should be extracted at realtime for feedbacks. Unfortunately, obtaining realtime cellular events on commodity phones is not readily available today. These events are not exposed to mobile OS or applications. There exist commercial tools (*e.g.*, QXDM [37]) and research projects (*e.g.*, LTEye [30]) to extract them. However, they require an external platform (*e.g.*, laptop or a special hardware (USRP)) to connect to the mobile device, which limits the device's flexible movement and its applicability. They cannot meet *iCellular*'s realtime requirements. To this end, we develop an in-phone solution MobileInsight [4] by exploiting the existing cellular diagnostic mode. We enable the diagnostic mode on the phone, modify the the virtual device for it (root access needed), and finally expose them to *iCellular*. This solution can be deployed on commodity phones without hardware changes.

5 Implementation

We have implemented *iCellular* on Motorola Nexus 6 and Huawei Nexus 6P. They run Android OS 5.1 and 6.0 using Qualcomm Snapdragon 805 and 810 chipsets, respectively. Both support 4G LTE, 3G HSPA/UMTS/CDMA and 2G GSM. To activate access to multiple cellular networks, we have installed *Project Fi* SIM card on Nexus 6/6P, which supports T-Mobile and Sprint 3G/4G. Figure 10 illustrates the system implementation. *iCellular* runs as a daemon service on a rooted phone. To enable interactions with the cellular interface, we activate the baseband processing tools (in bootloader), and turn on the diagnostic mode [2] and AT-command interfaces.

Basic APIs. *iCellular* allows the device to control its cellular access strategies through three APIs: `Monitor()` for active monitoring (§4.1), `Predictor()` for performance prediction (§4.3) and `SwitchTo()` for direct switching (§4.2). The decision fault tolerance is enabled by default (§4.4). Appendix A presents an illustrative example on how to use them.

Usability-Flexibility tradeoff. The above basic APIs provide most flexible means to customize access strategies. In practice, however, there is no need for most normal users to customize the strategies from the scratch. To support better usability, *iCellular* provides some built-in strategies on top of the basic APIs. Devices can choose these pre-defined ones, rather than build customized versions by themselves. We have developed three strategies: prediction-based, radio quality only and profile only (see §6 for performance comparisons).

Adaptive active monitoring (§4.1). We implement `Monitor()` with manual search and adaptations. Our prototype initiates the search with an AT query command `AT+COPS=?`. The non-disruption and minimal search adaptations are implemented for events of Table 1.

Adaptive direct switch (§4.2). We implement the `SwitchTo()` on top of PLMN selection, with dynamic adaptations for direct switch. Ideally, this can be executed with the AT command `AT+COPS=manual,carrier,network`. However, this command is forbidden by the cellular interface of Nexus 6/6P. We thus take an alternative approach. We modify the preferred network type through Android's API `setPreferredNetworkType`, and change the carrier with *Project Fi*'s secret code. Admittedly, this approach may incur extra switch overhead, but it is still acceptable (§6.2).

Prediction for heterogenous carriers (§4.3). We implement `Predictor()` in two steps. First, we implement the online sample collection, which collects radio measurements, RRC configurations and QoS profiles as features. We also define a callback to collect the network/application-level performance metrics. We then implement the online regression tree algorithm for training and prediction.

Decision fault prevention (§4.4). The fault prevention function is implemented as a shim layer between active monitoring and basic APIs. It detects the potential switch faults based on monitoring results and heterogeneity profiling, and excludes the unreachable carrier networks from the monitoring results. We further add a runtime checker in `SwitchTo()`, and prevent devices from selecting carriers not in the scanning results.

Cellular events collection (§4.5). We use the built-in realtime cellular loggers from *MobileInsight*. We develop a proxy daemon for the diagnostic port (/dev/diag), and redirect the events to the phone memory.

6 Evaluation

We evaluate *iCellular* along two dimensions. We first present the overall performance improvement by *iCellular* with smart multi-carrier access (§6.1), and then show *iCellular* satisfies various design properties in §4 (§6.2). All experiments are conducted on commodity Nexus 6 phones with *iCellular* in two cities of Los Angeles (west coast) and Columbus (Midwest), mainly around two campuses. The results on Nexsus 6P are similar.

6.1 Overall Performance

We use four representative applications to assess *iCellular*: SpeedTest (bulk file transfer), Web (interactive latency for small volume traffic), Youtube (video streaming) and Skype (realtime VoIP). We evaluate each application with quality-of-experience metrics whenever possible, *i.e.*, downlink speed for SpeedTest, page-loading time for Web [13] (measured with Firefox), video suspension time for Youtube [32] (measured by its APIs), and latency for Skype [27] (measured with its tech info panel). The details to collect application performance metrics are given in Appendix B. We run both pedestrian mobility and static tests. Along the walking routes, we uniformly sample locations. Note that *Project Fi*'s automatic selection protects the device's data connectivity by deferring its switch to the idle mode. For fair comparisons, we move to each sampled location in the idle mode (no voice/data, screen off), wait for sufficiently long time (≥1min) for potential switch during idle, and then start to test each application. We have at least five test runs and use the median value for evaluation.

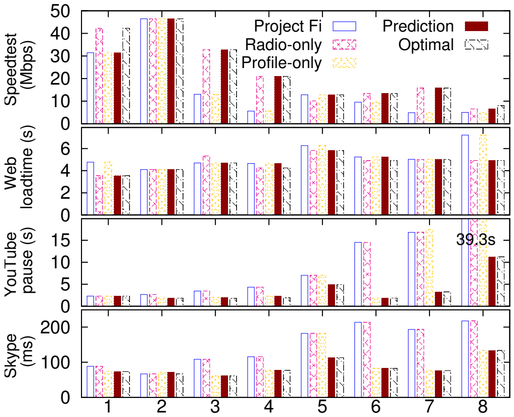


Figure 11: Performance of Speedtest, Web, Youtube, Skype using various multi-carrier access schemes.

We compare *iCellular* and its variants, with two baselines: *(i)* ***Project Fi*'s automatic selection** and *(ii)* **Optimal strategy:** We obtain the optimal access option by exhausting the application or network performance at each location. It may not be achieved in reality, but it serves as an ideal performance benchmark. We test three built-in *iCellular* decision strategies (§5): (1) **Prediction-based:** the default strategy in *iCellular*, which chooses the carrier with the best ranking metric from the predictor §4.3. The predictor is trained based on our one-month user-study logs, and tested over different routes. (2) **Radio-only**: the *de-facto* handoff strategy in 3G/4G. We implement the standardized cell re-selection scheme [11]. Whenever a network 4G with its signal strength higher than -110dBm (defined in [11]) exists, the strongest 4G carrier is chosen. Otherwise, we choose the strongest 3G network. (3) **Profile-only:** the device is migrated to the carrier network with the highest QoS (see Table 2). For our *iCellular* strategies, we use the carrier list with all network types supported by *Project Fi* (*i.e.*, 3G and 4G in T-Mobile and Sprint).

Figure 11 plots their performances in eight instances (locations), which belong to three categories: both carriers with acceptable coverage (Case 1-2), one carrier with acceptable coverage but the other not (Case 3-5), both carriers with weak coverage and one is even weaker (Case 6-8). We further compare them with the optimal one in two dimensions: accuracy toward the optimality, and the performance gap/improvement.

Accuracy toward optimality. We compare the probability that each scheme reaches the optimal network. Let I and I_{opt} be the access options chosen by the test scheme and the optimal strategy. We define the hit ratio as the matching samples $|(I \doteq I_{opt})|$ over all test samples. Table 3 shows the hit ratios of all schemes by different applications. *iCellular*'s prediction-based strategy

	Project Fi	Radio-only	Profile-only	Prediction
Speedtest	47.3%	63.1%	36.8%	73.6%
Web	57.9%	73.6%	31.6%	57.8%
Youtube	16.9%	22.6%	49.1%	50.9%
Skype	24.5%	7.6%	84.9%	92.5%

Table 3: Statistics of accuracy toward the optimality.

	SpeedTest	Web	Youtube	Skype
Radio meas	36.5%	72.7%	26.4%	8.7%
Heterogeneity profile	63.5%	27.3%	73.6%	91.3%

Table 4: Weights of radio measurement and network profiles in *iCellular*'s prediction strategy.

makes a wiser multi-carrier access decision. The hit ratios are 73.6%, 57.8%, 50.9% and 92.5% in SpeedTest, Web, Youtube and Skype, respectively. They are relatively small in Web and Youtube, but do not incur much performance degradation (explained later). They are usually higher than *Project Fi*'s automatic selection except for Web. The mobility speed has minor impact on the prediction accuracy, since it does not affect sample collection. Both radio measurements and cellular network profiles contribute to the high accuracy, but their impacts on all apps vary. We calculate their normalized variable importance in the regression tree (defined in [31]) and Table 4 shows their weights for four apps. We also find that, the metric specific for one app often locates the better network for other apps at the same location. The reason is that, the characteristics of one carrier network tend to have consistent impact on all apps. When the performance gap between two carriers is significant, it would exhibit on all application-level metrics.

Data service performance. We next examine the data performance by different schemes. We define the gap ratio $\gamma = |x - x^*|/x^*$, where x is the performance using various access strategies, x_{opt} is the optimal performance. We plot CDF of γ in Figure 12 and present the hit ratios and statistics of γ^+ in Table 5. Compared with *Project Fi*, *iCellular* narrows its performance gap (*e.g.*, reducing the maximal speed loss from 73.7% (19.7Mbps) to 25.7%, and the maximal video suspension time gap from 28.1s to 3.2s). The performance gain varies with locations (see Figure 11). With acceptable coverage (Case 1-2), *Project Fi*'s performance also approximates the optimal one. However, at locations with weak coverage, *iCellular* improves the device performance more visibly. The performance gain varies with applications (traffic patterns). Compared with other traffic, *iCellular* provides relatively small improvement for Web browsing. The reason is that, the Web traffic volume is relatively small, and no large performance distinction appears among various access options. However, for heavy traffic (e.g., file transfer), video streaming and voice calls, *iCellular* substantially improves the performance. The average improvement of *iCellular* over *Project Fi* approximates $\gamma_{fi} - \gamma_{icellular}$. On average, *iCellular* increases 23.8% downlink speed and reduces 7.3% loading time in Web, 37% suspension time in Youtube, 60.4% latency in Skype. Since *iCellular* often selects the optimal access, the maximal gain over *Project Fi* can be up to 46.5% in Web, 6.9x in Youtube, 1.9x in Skype, and 3.74x in Speedtest.

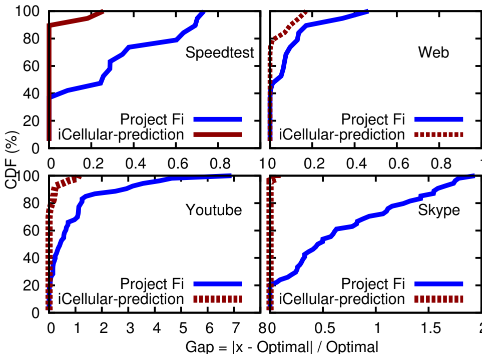


Figure 12: The performance gaps from *Project Fi* and *iCellular*'s prediction strategy to the optimality.

	Project Fi		iCellular-prediction	
	med(γ) ($\|x - x^*\|$)	max(γ) ($\|x - x^*\|$)	med(γ) ($\|x - x^*\|$)	max(γ) ($\|x - x^*\|$)
SpeedTest (speed)	36.2% 3.8Mbps	73.3% 19.8Mbps	**12.4%** 1.4Mbps	25.7% 9.8Mbps
Web (loadtime)	8.5% 0.5s	46.5% 2.3s	**1.2%** 0.2s	17% 0.7s
Youtube (Pause)	55% 1.4s	690% 28.1s	**18%** 0.3s	111% 3.2s
Skype (Latency)	62.9% 64ms	193.8% 117ms	**2.5%** 4.4ms	6.7% 4.5ms

Table 5: Performance gaps from the optimal one.

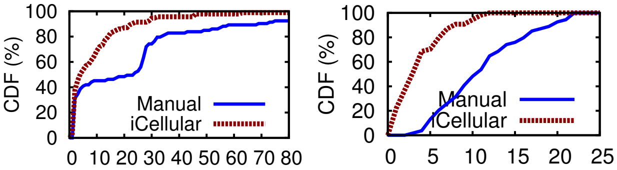
Comparison between *iCellular*'s built-in strategies. *iCellular*'s prediction strategy best approximates the optimal strategy. It outperforms radio-only and profile-only variants (§4.3). We also see that, the importance of profile and radio measurements varies across applications. For example, our log-event analysis shows that, T-Mobile assigns *Project Fi* devices to the interactive traffic class (Table 2), which is optimized for delay-sensitive service [6][1]. Instead, Sprint only allocates the best-effort traffic class to these devices. This explains why the profile-only strategy's performance approximates the optimal strategy for Skype. It also implies that, for a given application (*e.g.*, Skype), simpler strategy (rather than prediction), which incurs smaller system overhead, can be available for close-to-optimal performance.

6.2 Efficiency and Low Overhead

We next present the micro-benchmark evaluations on *iCellular*'s key components, and validate that they are efficient. We examine the active monitoring, direct switch and fault prevention, as well as the overhead of signaling, CPU, memory and battery usage.

Efficiency. We examine *iCellular*'s efficiency through two adaptive module tests. First, we show that, *iCel-*

[1] This QoS is specific to *Project Fi*. For example, we verify that a T-Mobile device with Samsung S5 is assigned lower background class.



(a) Total search time (b) Cell count

Figure 13: *iCellular*'s adaptive monitoring avoids exhaustive search.

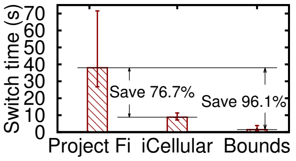


Figure 14: Inter-carrier switch time.

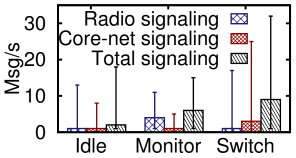
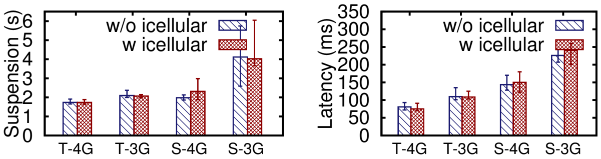


Figure 15: Cellular signaling overhead.

lular's adaptive monitoring is able to accelerate carrier scanning. We compare it with the default manual search, and record the total search time and the number of cells scanned at 100 different locations. Figure 13 shows that, with adaptive search, 70% of the complete search can be completed within 10s, 64% shorter than the exhaustive manual search. Note that devices are allowed to switch before the complete search (§5), so it waits shorter in practice. Figure 13b counts the scanned cells, and validates that such savings come from avoiding those unnecessary cell scans. The search time and the number of cells vary with locations and the cell density.

Second, we examine how well *iCellular*'s adaptive switch reduces service disruption. In this experiment, we place the phone at the border of two carriers' coverages, and test the switch time needed for *iCellular* and *Project Fi* for 50 runs. The inter-carrier switch time is defined as the duration from the de-registration from the old carrier to the registration to the new carrier. For comparison purposes, we also calculate the lower bound based on the *MobileInsight* event logs, described in §4.2. Figure 14 shows that, *iCellular* saves 76.7% switch time on average, compared with *Project Fi*. However, the current *iCellular* prototype has not achieved the minimal switch time: it still requires 8.8s on average. Under high-speed mobility, this may delay the switch to the optimal carrier network. We dig into the event logs, and discover that, the current bottleneck lies in the SIM card reconfiguration. The current *iCellular* implementation relies on *Project Fi*'s system service. It has to wait until the SIM card is reconfigured to switch to another carrier. In the experiments, we find that most of the switch times (7.3s on average) are spent on the SIM card reconfiguration, which is beyond the control of *iCellular*. The phone has no network service in this period. The lower bound implies that, with better SIM card implementation, *iCellular* could save up to 96.1% of switch time compared with the *Project Fi*.



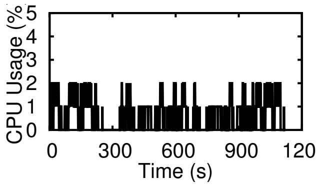
(a) Youtube (b) Skype

Figure 16: *iCellular*'s active monitoring has minor impacts on data performance.

Fault prevention. We next verify that *iCellular* handles fault scenarios and prevents devices from switching to unwise carrier networks. All three failure types in §4.4 have been observed in our one-month user study. Note that the failure scenarios are not very common in reality. We observe one instance of the forbidden access, where a Sprint 4G base station sets the access-barring option for 10 min (possibly under maintenance). We observe another instance of Figure 8, where T-mobile 4G is available but T-Mobile 3G is not available. Since T-Mobile 4G does not provide Voice over LTE (VoLTE) to *Project Fi* and has to rely on its 3G network (using circuit-switching Fallback) for voice calls [41]. Consequently, the correct decision should be to not switch to T-Mobile 4G, since voice calls are not reachable there. *iCellular* detects it from the profiled call preference and location update messages, and excludes this access option from the candidate list. We also observe uncoordinated mobility rules between the network and the device (Figure 9). We validate that *iCellular* can detect and avoid them.

Impact on applications in monitoring. We show that *iCellular*'s active monitor does not disrupt the ongoing data service at the device. We run the active monitor 100 times with/without applications and its active data transfer. We test with four applications and the results with-/without *iCellular*'s monitoring are similar. Figure 16 shows the performance with/without *iCellular*'s monitoring for Youtube and Skype. Enabling/disabling active monitoring has comparable application performance. As explained in §4.1, this is because the carrier scanning procedure is performed only in the absence of traffic.

Signaling overhead. We show that *iCellular* incurs moderate signaling messages to the device and the network. We record the device-side signaling message rate under three conditions (when running our performance tests): (1) *Idle:* No monitoring/switch functions are active. No extra cellular signaling messages are generated; (2) *Monitor: iCellular* initiates its active monitoring. The device should receive more broadcasted signals. However, no extra signaling messages are generated to the network; (3) *Switch: iCellular* initiates the switch to the new carrier network. Because of the registration, extra signaling messages are generated to both the device and the network. For all scenarios, we count the radio-



(a) CPU usage

WeChat	21.03%
iCellular	**4.75%**
Skype:	4.54%
Gmail:	3.33%
GO Power Master	1.21%
Google Play	0.73%
...	

(b) Battery usage

Figure 17: CPU and battery usage of *iCellular*.

level (from RRC layer), core-network level (from mobility and session management layers) and the total signaling rate. Figure 15 shows that, the maximum observed signaling message rate is 32 message/sec.

CPU and memory. In all our tests, the maximum CPU utilization is below 2%, while the maximum memory usage is below 20 MB (including virtual memory). Figure 17a shows a 20-min log during a driving test, where its maximum memory usage is 16.45MB.

Energy consumption. Since we cannot directly measure the consumed power at Nexus 6/6P with an external power meter (its battery is sealed, and hard to remove), we take an application-level approach. We use a fully-charged Nexus 6 phone and run it for 24 hours. We use an app called GO-Power-Master [3] to record energy consumption for each component/app. Figure 17b shows one record, where *iCellular* explicitly consumes about 4.75% of battery. Its energy can be further optimized (*e.g.*, with sleep mode and periodical monitoring).

7 Discussion

Working with network-side solution. Despite a device-side solution, *iCellular* can work in concert with carrier-side mechanisms for better performance. For example, during the inter-carrier switch, *iCellular* could benefit from the network-side buffering and tunneling of downlink traffic, and support more seamless migration. For each individual carrier, its network-side solution can also benefit from *iCellular* with device-side feedbacks on all available carriers. Note that the carrier network still retains its final say on the switch decision by rejecting the device-initiated switch requests.

Hints for future mobile network design. The future multi-carrier access design (*e.g.*, 5G) can benefit from our *iCellular*'s design experience. For example, the idea of adaptive monitoring (§4.1) and direct switch (§4.2) may be instrumental to designing a new inter-carrier switch mechanism beyond the popular PLMN selection. The heterogeneity predictor (§4.3) and decision fault prevention (§4.4) are also directly applicable to 5G.

8 Related Work

In recent years, exploiting multiple cellular carriers attracts research efforts on both network and device sides. The network-side efforts include sharing the radio resource [22, 28, 36, 36] and infrastructure [17, 18, 29, 44] between carriers, which helps to reduce deployment cost. On the device side, both clean-slate design with dual SIM cards [1,20] and single universal SIM card [14,24,26] are used for multi-carrier access. But multi-SIM phones provide multi-carrier access in a constrained fashion. The number of accessible carriers is limited by the number of SIM cards (usually two due to energy and radio interference constraints). Our work complements the single-SIM approach for incremental deployment. It differs from existing efforts by leveraging low-level cellular information, and offering device-defined selections in a responsive and non-disruptive manner.

iCellular leverages the rich cellular connectivity on the device. Similar efforts use multiple physical interfaces from WiFi and cellular, including WiFi offloading [16,21,23] and multipath-TCP [35,43]. *iCellular* differs from all these in that it still uses a single cellular interface. [15] reports similar problems, but we further unveil their root causes. Similar issues may also occur with traditional handoffs within a single carrier [39, 40, 45], which are caused by the carrier's own problematic management. Instead, *iCellular* targets inter-carrier migration, and chooses to let end devices customize the selection strategies among carriers.

9 Conclusion

The current design of cellular networks limits the device's ability to fully explore multi-carrier access. The fundamental problem is that, existing 3G/4G mobile networks place most decisions and operational complexity on the infrastructure side. This network-centric design is partly inherited from the legacy telecom-based architecture paradigm. As a result, the increasing capability of user devices is not properly exploited. In the multi-carrier access context, devices may suffer from low-quality access while incurring unnecessary service disruption. In this work, we describe *iCellular*, which seeks to leverage the fine-grained cellular information and the available mechanism at the device. It thus dynamically selects better mobile carrier through adaptive monitoring and online learning. Our initial evaluation validates the feasibility of this approach.

Acknowledgments: We thank the anonymous reviewers for their constructive comments. We greatly appreciate our shepherd, Dr. Matt Welsh for his continuous and timely guidance to improve the work. We also appreciate Dr. Richard Liu and the Project Fi team for their valuable feedback. This work is supported in part by NSF awards (CNS-1423576, CNS-1421440, CNS-1526456, and CNS-1526985).

References

[1] Dual sim phone. https://en.wikipedia.org/wiki/Dual_SIM.

[2] Enabling diagnostic mode in mobileinsight. http://metro.cs.ucla.edu/mobile_insight/diag_reference.html.

[3] Go power master. https://play.google.com/store/apps/details?id=com.gau.go.launcherex.gowidget.gopowermaster&hl=en.

[4] Mobileinsight project. http://metro.cs.ucla.edu/mobile_insight.

[5] 3GPP. Lte ue category. http://www.3gpp.org/keywords-acronyms/1612-ue-category.

[6] 3GPP. TS23.107: Quality of Service (QoS) concept and architecture.

[7] 3GPP. TS27.007: AT command set for User Equipment (UE), 2011.

[8] 3GPP. TS25.304: User Equipment (UE) Procedures in Idle Mode and Procedures for Cell Reselection in Connected Mode, 2012.

[9] 3GPP. TS36.331: Radio Resource Control (RRC), 2012.

[10] 3GPP. TS24.301: Non-Access-Stratum (NAS) for EPS; , Jun. 2013.

[11] 3GPP. TS36.304: User Equipment Procedures in Idle Mode, 2013.

[12] 3GPP. TS23.122: Non-Access-Stratum (NAS) functions related to Mobile Station (MS) in idle mode, 2015.

[13] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google‘s Data Compression Proxy for the Mobile Web. In *USENIX NSDI*, 2015.

[14] Apple. Apple SIM for iPad. https://www.apple.com/ipad/apple-sim/.

[15] N. Armstrong. Network handover in google fi, 2015. http://nicholasarmstrong.com/2015/08/network-handover-google-fi/.

[16] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting Mobile 3G Using WiFi. In *ACM MobiSys*, 2010.

[17] R. Copeland and N. Crespi. Resolving ten MVNO issues with EPS architecture, VoLTE and advanced policy server. In *IEEE International Conference on Intelligence in Next Generation Networks (ICIN)*, pages 29–34, 2011.

[18] X. Costa-Pérez, J. Swetina, T. Guo, R. Mahindra, and S. Rangarajan. Radio Access Network Virtualization for Future Mobile Carrier Networks. *IEEE Communications Magazine*, 51(7):27–35, 2013.

[19] S. L. Crawford. Extensions to the cart algorithm. *International Journal of Man-Machine Studies*, 31(2):197–217, 1989.

[20] S. Deb, K. Nagaraj, and V. Srinivasan. MOTA: Engineering an Operator Agnostic Mobile Service. In *ACM MobiCom*, pages 133–144, 2011.

[21] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. WiFi, LTE, or Both? Measuring Multi-Homed Wireless Internet Performance. In *ACM IMC*, pages 181–194, 2014.

[22] P. Di Francesco, F. Malandrino, and L. A. DaSilva. Mobile Network Sharing Between Operators: A Demand Trace-driven Study. In *ACM CSWS*, 2014.

[23] S. Dimatteo, P. Hui, B. Han, and V. O. K. Li. Cellular Traffic Offloading through WiFi Networks. In *IEEE MASS*, 2011.

[24] Engadget. Apple and Samsung in talks to adopt e-SIM technology. http://www.engadget.com/2015/07/16/apple-samsung-e-sim/.

[25] Google. Youtube android player api. https://developers.google.com/youtube/android/player/reference/com/google/android/youtube/player/YouTubePlayer.

[26] Google. Project fi, 2015. https://fi.google.com/about/.

[27] S. Jelassi, G. Rubino, H. Melvin, H. Youssef, and G. Pujolle. Quality of Experience of VoIP Service: A Survey of Assessment Approaches and Open Issues. *IEEE Communications Surveys & Tutorials*, 14(2):491–513, 2012.

[28] M. Jokinen, M. Mäkeläinen, and T. Hänninen. Demo: Coprimary Spectrum Sharing with Inter-operator D2D Trial. In *ACM MobiCom*, 2014.

[29] R. Kokku, R. Mahindra, H. Zhang, and S. Rangarajan. NVS: A Substrate for Virtualizing Wireless Resources in Cellular Networks. *IEEE/ACM Transactions on Networking (TON)*, 20(5):1333–1346, 2012.

[30] S. Kumar, E. Hamed, D. Katabi, and L. Erran Li. LTE Radio Analytics Made Easy and Accessible. In *ACM SIGCOMM*, pages 211–222, 2014.

[31] MathWorks. Variable importance in regression tree. http://www.mathworks.com/help/stats/compactregressiontree.predictorimportance.html.

[32] R. K. Mok, E. W. Chan, and R. K. Chang. Measuring the Quality of Experience of HTTP Video Streaming. In *IFIP/IEEE Integrated Network Management (IM)*, pages 485–492, 2011.

[33] Mozilla. Remotely debugging firefox for android. https://developer.mozilla.org/en-US/docs/Tools/Remote_Debugging/Firefox_for_Android.

[34] NGMN. Ngmn 5g white paper. https://www.ngmn.org/work-programme/5g-initiative/.

[35] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure. Exploring Mobile/WiFi Handover with Multipath TCP. In *ACM CellNet*, 2012.

[36] J. S. Panchal, R. Yates, and M. M. Buddhikot. Mobile Network Resource Sharing Options: Performance Comparisons. *IEEE Transactions on Wireless Communications*, 12(9):4470–4482, 2013.

[37] Qualcomm. QxDM Professional - QUALCOMM eXtensible Diagnostic Monitor. http://www.qualcomm.com/media/documents/tags/qxdm.

[38] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Discovering Fine-grained RRC State Dynamics and Performance Impacts in Cellular Networks. In *ACM MobiCom*, pages 177–188, 2014.

[39] A. Salkintzis, M. Hammer, I. Tanaka, and C. Wong. Voice Call Handover Mechanisms in Next-Generation 3GPP Systems. *Communications Magazine, IEEE*, 47(2):46–56, 2009.

[40] K. E. Suleiman, A.-E. M. Taha, and H. S. Hassanein. Understanding the interactions of handover-related self-organization schemes. In *Proceedings of the 17th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '14, pages 285–294, 2014.

[41] G. Tu, C. Peng, H. Wang, C. Li, and S. Lu. How Voice Calls Affect Data in Operational LTE Networks. In *MobiCom*, Oct. 2013.

[42] G.-H. Tu, C. Peng, C.-Y. Li, X. Ma, H. Wang, T. Wang, and S. Lu. Accounting for Roaming Users on Mobile Data Access: Issues and Root Causes. In *ACM MobiSys*, 2013.

[43] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *USENIX NSDI*, 2011.

[44] Y. Zaki, L. Zhao, C. Goerg, and A. Timm-Giel. Lte mobile network virtualization. *Springer Mobile Networks and Applications*, 16(4):424–432, 2011.

[45] H. Zhang, X. Wen, B. Wang, W. Zheng, and Y. Sun. A novel handover mechanism between femtocell and macrocell for lte based networks. In *Proceedings of the 2010 Second International Conference on Communication Software and Networks (ICCSN)*, 2010.

Appendices

A An Example of iCellular APIs

We use a simple example to illustrate how they work. Consider a device who has access to T-Mobile and Sprint 3G/4G networks, and would like to choose in the network with minimal radio link latency. To do so, the device first initiates an *active monitor*, and specifies the list of the carrier networks s/he is interested in:

```
monitor = Monitor(["T-4G","T-3G","S-4G"]);
```

To choose the target carrier network, the user may want to learn each network's performance. The following code shows how user can initiate a latency predictor:

```
predictor = Predictor("Latency");
```

To let devices make responsive decisions, *iCellular* let selection strategy be triggered by the latest and even partial search results. To do so, the device should overload an *event-driven* decision callback function. Devices are given the runtime monitoring results of available carrier networks. Optionally, the device can use the predictor to help determine the target carrier network. The device can call `SwitchTo()` function to perform the switch. The following code shows a strategy that minimizes latency:

```
def decision_callback(monitor):
  min_latency = inf; target = null;
  for network in monitor:
    latency = predictor.predict(network);
    if latency < min_latency:
      min_latency = latency;
      target = network;
  SwitchTo(target);
```

B Collecting App-specific Performance

For SpeedTest, we directly record the downlink speed for each test. Note that Nexus 6 supports LTE category 4, which can yield up to 150Mbps downlink bandwidth in theory [5]. This is why we observe 40+Mbps downlink speed in our tests, which is much higher than most previous measurements. For Web, currently we use Firefox and get the web loading time from its debugging console [33]. For Youtube, we extract its buffering time by tracking `OnBuffer(True)` and `OnBuffer(False)` events from Youtube Android player API [25], and calculating the elapsed time in between, during which the user has to pause the video. For Skype, we collect round-trip latencies (in ms) as the performance metric. To get it, We enabled the `Technical info` panel in the Skype app, which shows the latency in the call. Then we record the round-trip latency in every second.

Diamond: Nesting the Data Center Network with Wireless Rings in 3D Space

Yong Cui[1], Shihan Xiao[1], Xin Wang[2], Zhenjie Yang[1], Chao Zhu[1], Xiangyang Li[1,3], Liu Yang[4], and Ning Ge[1]

[1]Tsinghua University
[2]Stony Brook University
[3]University of Science and Technology of China
[4]Beijing University of Posts and Telecommunications

Abstract

The introduction of wireless transmissions into the data center has been shown to be promising in improving the performance of data center networks (DCN) cost effectively. For high transmission flexibility and performance, a fundamental challenge is to increase the wireless availability and enable fully hybrid and seamless transmissions over both wired and wireless DCN components. Rather than limiting the number of wireless radios by the size of top-of-rack (ToR) switches, we propose a novel DCN architecture, *Diamond*, which nests the wired DCN with radios equipped on all servers. To harvest the gain allowed by the rich reconfigurable wireless resources, we propose the low-cost deployment of scalable 3D Ring Reflection Spaces (RRSs) which are interconnected with streamlined wired herringbone to enable large number of concurrent wireless transmissions through high-performance multi-reflection of radio signals over metal. To increase the number of concurrent wireless transmissions within each RRS, we propose a precise reflection method to reduce the wireless interference. We build a 60GHz-based testbed to demonstrate the function and transmission ability of our proposed architecture. We further perform extensive simulations to show the significant performance gain of Diamond, in supporting up to five times higher server-to-server capacity, enabling network-wide load balancing, and ensuring high fault tolerance.

1 Introduction

The high-performance data center network (DCN) is an essential infrastructure for cloud computing. There is a quick growth of large-scale services (e.g., Google Search, Hadoop, MapReduce, etc.) in the cloud, and recent measurements show tremendous traffic variations over space and time in DCNs [5, 7, 8, 20]. Conventional wired DCNs generally adopt the fixed and symmetric network design. This may lead to prevalent hot spots across different layers of the architecture which significantly reduces the performance of DCNs [7, 20, 37].

There are some recent interests on constructing *hybrid* DCNs [18, 19, 33, 38, 39] with the introduction of new network components such as optical circuit switches or wireless radios into the DCNs to provide configurable links [9,13,25,28]. Although these hybrid infrastructures show the potential in achieving higher DCN capacity and lower transmission delay, their wired structures are kept unchanged even though they are not primitively designed to work with new network techniques, which limits the performance of hybrid DCN. Specifically, the new network components are added directly into conventional DCNs or are applied to replace part of existing network switches [18, 33, 38, 39]. Considering only the *local* performance improvement, it is hard for existing schemes to achieve the global optimal performance in the presence of network-wide traffic changes. The key challenge of a *fully hybrid* network design is to form a novel hybrid network architecture that can take full advantage of different network techniques and enable coherent and seamless transmissions for much higher DCN performance.

The low cost of today's commodity 60GHz radios makes their wide deployment a better option in a fully hybrid network design [39]. Providing high wireless availability in the data center is the key to achieving high performance gain in a hybrid architecture. In existing proposals for hybrid DCNs, wireless radios are generally deployed on a flat 2D plane at the top of racks, which is susceptible to signal blocking [38]. Although a flat reflector on the room ceiling was proposed to alleviate the problem [19, 38], the ceiling height is quite restricted (3 meters [38]) and the method requires clearance above racks, which is usually infeasible in conventional data centers. The small rack size also restricts the number of radios that can be placed on each rack (at most eight radios per rack [38, 39]). If radios are densely deployed on top of racks, the strong inter-ratio interference would restrict the number of concurrent wireless links thus constraining the system performance [39]. The need of deploying more radios and links in the hybrid network for higher wireless availability calls for a completely new DCN architecture design.

In this work, we propose a novel *fully-hybrid* network architecture, named Diamond, which ensures high wireless availability for efficient and high-performance DCN communications. Rather than restricting the radios to be on top of racks, we propose to deploy wireless radios along with a large number of servers. To avoid the interference among dense radios at the 2D plane, we propose to construct multiple **R**ing **R**eflection **S**paces (RRSs) to make the radios sparsely distributed in the 3D space. Inside each RRS, we develop a novel multi-reflection method to address the blocking problem on building wireless links. With our design, there is no need of changing the room plan above racks. Diamond has three key design features:

Novel hybrid network topology (§2): Rather than adding wireless radios directly on top of racks, we propose a fully hybrid network topology by constructing RRSs in Diamond to facilitate wireless transmissions and isolate the wireless interference. It also supports direct server-to-server wireless links rather than conventional rack-to-rack links. Then we apply a streamlined wired herringbone to interconnect the RRSs at low cost.

Precise multi-reflection of wireless links (§3): The susceptibility to blocking and the interference are two major issues that limit the wireless performance in DCNs. To the best of our knowledge, this is the first work that develops the multi-reflection transmission method to address the challenge of signal blocking. We further design a novel precise reflection scheme to efficiently restrict the wireless interference in the presence of a large number of concurrent wireless links.

Wireless & wired hybrid routing (§4): We propose an opportunistic hybrid routing scheme to allow for low transmissions delay and graceful fault tolerance. We further show that the network diameter of Diamond can scale logarithmically with the server number to effectively bound the route length.

We implement a 60GHz-based testbed, and our experimental results confirm the high performance of multi-reflection, and demonstrate that proper reflection holes can efficiently reduce the interference in 3D space (§6). Driven by the testbed parameters, our simulations show that Diamond can support up to five times higher server-to-server capacity and ensure graceful fault tolerance (§7). Finally, we introduce the related work (§8) and draw the conclusions (§9).

2 Architecture

In this section, we first introduce the basic architecture and methodologies used in the Diamond system, and then present its hybrid topology design.

2.1 Diamond system overview

At a high level, the Diamond system should meet the data center needs at different timescales. First, the configuration of wireless links should be updated periodically so that the network topology can better accommodate the current traffic of the data center. Second, given a configured network, we need to efficiently route the flows in real time.

Dynamic wireless configuration: Following the prior studies, the Diamond system exploits the controller of software-defined networking (SDN) for flexible and efficient configuration of the wireless links and routing paths [4,11,26,27]. More specifically, the Diamond controller periodically updates the configuration of the wireless links based on the traffic conditions reported from SDN-controllable ToR switches. Servers are equipped with high-capacity wireless radios (60GHz radio [18] or FSO transceivers [19]). To dynamically configure the wireless links, they are allowed to communicate with each other either directly by steering and aligning the antennas (physically or electronically driven [18,19]) or using a multi-reflection method we propose. The controller first builds wireless links to alleviate the heavy traffic from the hot spots, and then randomly forms additional wireless links using the remaining available radios to achieve the benefits of random networking [31].

Hybrid routing: The controller only computes the routing paths of hot-spot server pairs during the wireless configuration to alleviate the hot-spot traffic globally for the network-wide load balancing thus higher network throughput. For other light-loaded server pairs, the routing decision is made distributedly by servers and switches so that their traffic can go through available wireless links opportunistically to cut short the routing paths in real time.

2.2 Key methodologies

There are two main challenges to implement a fully hybrid network: (1) When a large number of wireless links are enabled, the interference will restrict the number of concurrent transmissions; and (2) When a large number of wireless radios are deployed, the high-frequency wireless links are easily blocked by obstacles such as the supply pipes of air conditioning or the steel structures above racks. In light of these problems, Diamond applies a 3D deployment of the wireless radios to facilitate high number of concurrent wireless transmissions taking advantage of the following key techniques:

Space division multiplexing: To disperse the wireless radios, the radios in Diamond are installed with servers at different heights. Rather than deploying the wireless radios densely on only one flat 2D plane, we place the wireless radios on several separated large annular surfaces. Thus the deployment density of wireless radios is much lower than that of previous studies [38, 39]. The adjacent annular surfaces form a RRS where the signal can run from one radio to another. Due to the space division, the same set of wireless channels can be multiplexed across different RRSs, which helps isolate the interference in Diamond.

Multi-reflection transmission: Although more radios can be deployed in a 3D space, many radios cannot reach each other with existing direct point-to-point transmission or the one-reflection transmission [18, 19, 38, 39] due to the obstacle blocking. Instead, in Diamond, we utilize multiple reflections to bounce the signal emitted from one server to another. This helps to greatly increase the number of available wireless links. Following the prior work [38], our testbed experiment confirms that using the flat metal board as reflector can offer very good specular reflection with little energy loss or changing path loss during each reflection. This avoids the overhead of buffering and switching packets in multiple hops over intermediate switches.

Different types of directional antennas may have different beam widths [18]. For a multi-reflection path, there is a tradeoff between the antenna beam width and the tolerance for the antenna alignment error. The narrower the beam width, the higher the antenna gains, but the less the alignment error tolerance. In the extreme case of using FSO with nearly zero beam width, previous study shows that using electrically-driven Galvo mirrors is possible to implement precise steering control [19]. For conventional 60GHz antennas, the electrically-driven antenna array is promising to satisfy this requirement [18, 39].

Precise reflection technology: Since the wireless antenna may have a wide beam width [18], multiple reflections would introduce unexpected interference inside the 3D space due to the signal leakage of the beam (e.g., the undesired side lobes of the 60GHz wireless beam [18, 29]). In order to efficiently restrict and control the interference caused by reflections, we develop a precise reflection method with the careful placement of absorbing materials on the reflection boards. Most areas of the board are covered by absorbing paper while small holes are left so that only the intended signal reflections are made by hitting the hole, which leads to very little signal leakage.

2.3 Topology design

The basic *motivation* of the Diamond topology design is to enable more concurrent wireless transmissions. In our design, we separate the specific transmission functions of wireless and wired links in the network, so that both their distinct advantages on the transmission can be fully explored. We construct a ring-shape basic structure that enables wireless-only transmissions inside the ring employing the multi-reflections (§2.2). Then we apply the stable wired links to address the transmissions across different ring structures.

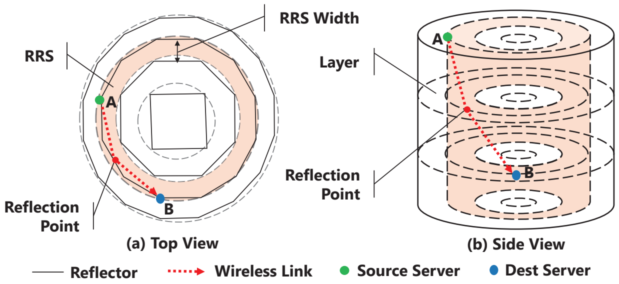


Figure 1: Brief view of the wireless ring in Diamond (N=4 rings and H=4 layers)

From the top view in Fig. 1, Diamond's topology is constructed by several concentric regular polygons with increasing radius. Polygons are numbered from inside to outside and named by *rings*, i.e., $\{R_i\}$, $1 \leq i \leq N$, where N is the total number of polygons. The i_{th} ring has $4i$ edges. The racks are placed at the vertex points of each ring, and there are totally $\sum_{1 \leq i \leq N}(4i) = 2(N^2+N)$ racks, while flat metal reflectors are put at the edge of each ring. Rather than mounting the reflectors [19,38] on the ceiling, reflectors in Diamond stand in perpendicular to the ground and have the same height as that of racks, which avoids the need of using clear ceiling space for wireless transmissions in data centers. In the following, we introduce the designs of major Diamond components.

Server and rack. Each rack holds multiple servers at different height. The servers inside different racks at the same height form a *layer*, and the layers are numbered from the top to the bottom as $\{l_j\}$, $1 \leq j \leq H$. The height of each layer equals the height of a server at conventional racks, and the number of layers H equals to the number of servers in one rack. Therefore, a Diamond topology can accommodate totally $2(N^2+N)H$ servers. Each server is equipped with 1 Ethernet port and 2 wireless ports with directional antennas. The networking principles in Diamond are: (1) the links between two servers are wireless; (2) the links between a server and its ToR switch or between two ToR switches are wired.

Wireless links: The 3D space between two neighboring rings is called an RRS. For each server, one of its antennas points to RRS at its inner side and the other points to RRS at its outer side. By adjusting the antenna directions in the RRS, each server at ring R_i can flexibly communicate with other servers at different heights on rings R_i, R_{i-1} and R_{i+1} through direct transmissions or multiple reflections on different reflectors (Fig. 1 and Fig. 3).

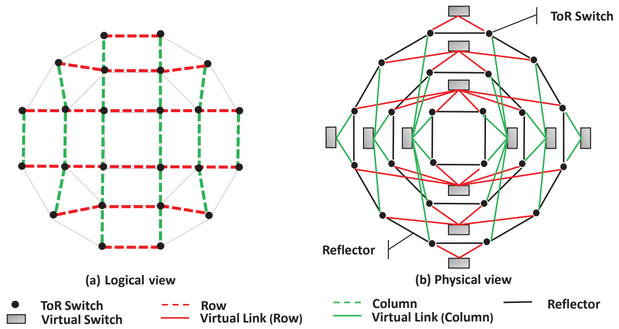


Figure 2: Top view of the wired herringbone of Diamond (N=3 rings)

Wired links: With wireless links formed locally inside each RRS, the wired links are applied to interconnect different RRSs. Fig. 2 gives a top view of the wired connections in Diamond. Similar to conventional DCN-s, the servers on each rack are connected to the common ToR switch. Fig. 2(a) shows the logical view of the *wired herringbone*. We number the horizontal lines in Fig. 2(a) from the top to the bottom as rows $\{r_i\}$, $1 \leq i \leq 2N$, and number the vertical lines from the left to the right as columns $\{c_i\}$, $1 \leq i \leq 2N$. Fig. 2(b) shows the physical connections of the wired herringbone. The principle of Diamond to interconnect the RRSs is that the ToR switches on the same row (or column) are interconnected by a *virtual switch*, while the ToR switches on different rows and different columns are not directly connected.

To implement the function of virtual switch, we have the option of applying any existing structure, e.g., the tree-based structure (Fat-tree [3]) or cube-based structure (BCube [16]), to interconnect the ToR switches on each row and each column. These structures may make the wired design of Diamond complex and costly. In Diamond, we prefer to apply the de-Bruijn graph [12] so that no additional switches are required. De-bruijn is attractive for providing constant link degree at each node and logarithmic network diameter. Then the path length is bounded and the routing structure is still simple (§4). Although using de-Bruijn structure often involves complex wiring [17], the wiring is kept simple in Diamond because only one row (or column) of ToR switches are connected as one de-Bruijn.

2.4 Rack and reflector arrangement

There are two requirements to arrange the racks in Diamond to facilitate its practical and scalable deployment: first, all the reflector boards should be flat and have the same length to facilitate their economical production; second, the RRS width should be kept stable, with the RRS width close to a fixed value when the number of rings increases. We call the physical distance between two neighboring rings R_i and R_{i+1} as the RRS width Δ_i (Fig. 1). Too large a RRS width will make Diamond occupy too much room area, while too small a RRS width will not leave enough space for wireless transmissions.

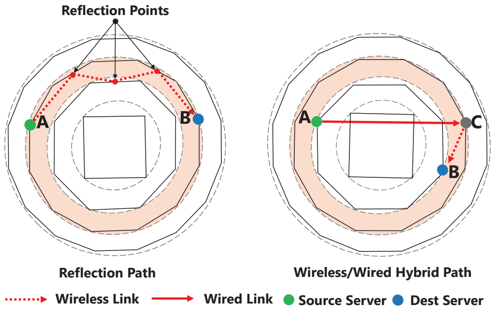


Figure 3: Routing in the 3D wireless ring

As mentioned earlier, all the polygons in Diamond are regular with the same edge length and are put concentrically in a symmetric way as shown in Fig. 2. The reflector height equals the height of racks, and the reflector length is denoted as L. Then our design ensures the RRS width Δ_i at ith ring to have the following property:

Property 1. $\lim\limits_{i\to\infty} \Delta_i = 2L/\pi$

Proof. *Based on the topology of Diamond, the radius d_i of ring R_i is $d_i = (\cot \frac{\pi}{4i})\frac{L}{2}$. Then we have $\Delta_i = d_{i+1} - d_i = (\cot \frac{\pi}{4(i+1)} - \cot \frac{\pi}{4i})\frac{L}{2} = \frac{\sin(\frac{\pi}{4}\frac{1}{i(i+1)})}{\sin[(\frac{\pi}{4})^2 \frac{1}{i(i+1)}]}\frac{L}{2}$. Hence we have $\lim\limits_{i\to\infty} \Delta_i = 2L/\pi$.*

Based on the above proof, the RRS width Δ_i decreases as the ring number i becomes larger. Property 1 ensures that the RRS width does not fall to zero but reaches a fixed limit value. For a setting L=2.5m, the RRS width Δ_i can keep a value close to the fixed limit value 1.6m. We can see that the RRS width becomes stable and approaches the fixed limit value quickly when the ring number increases, which demonstrates the scalability of the Diamond design.

3 Wireless configuration

In this section, we first introduce our schemes of finding the reflection path when building a wireless link and eliminating the wireless interference during the reflections, and then present our strategies in forming flexible wireless configurations for network-wide load balancing.

3.1 Reflection path

Since the physical topology of Diamond is fixed, the reflection paths can be easily calculated between any two servers. The calculation of the reflection path table is done offline at the initial deployment of Diamond. If there are multiple paths available between two servers, we choose the one with the least number of reflection

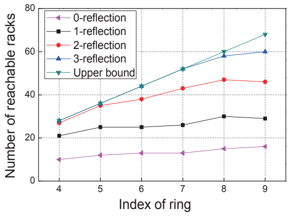


Figure 4: Number of reachable racks per server at different rings and within different reflection times

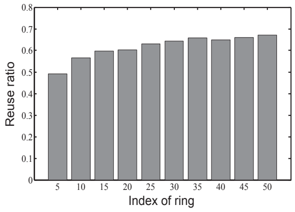


Figure 5: Reuse ratio of reflection points on a board at different rings

times (direct transmission is considered as zero times of reflection). Given a source server and a destination server in Diamond, if a reflection path can be found in the table, the antenna angles can be adjusted by the servers according to the table values to build the wireless link.

Based on the Diamond topology, we simulate the reflection paths between all the server pairs. Fig. 4 shows the average communication range of a wireless radio, i.e., the reachable rack number at both its current ring R_i and inner ring R_{i-1}. We can see that no more than three reflections can cover above 90% racks in the RRS when the ring number is less than 9. For a ring number larger than 4, a server can reach at least 10 racks through the direct transmission, 20 racks within a single reflection and 28 racks within two reflections.

3.2 Reduction of wireless interference

We design a precise reflection method to alleviate the wireless interference during reflections. Specifically, we carefully place the absorbing materials on the reflection board and leave small *holes* for only the intended reflection points. In the following, we analyze the density and distribution of reflection points (i.e., the reflection holes) on the reflector boards.

To simplify the analysis, we first present a special *circle case* where the flat reflectors are replaced by curved reflectors so that all the polygons are transformed to their circumcircles. We consider the communication of servers inside the kth RRS, i.e., the communication between a server on ring k and another on ring $k+1$ and the communication between two servers on ring $k+1$. The reflection times are limited within three. The communication of servers in different rings is achieved by zero and double reflections. The double reflection forms the reflection points on the outer side of ring k and the inner side of ring $k+1$.

Considering the distribution of reflection points on ring k, we have the following property:

Property 2. *At each layer of Diamond, for an arbitrary reflector on ring k, there are at most six reflection points on the reflector board.*

Proof. *Based on the coordinates of a server n on ring* $k+1$ *and a server m on ring k, we obtain the central angle for each reflection point, denoted as* $\frac{2}{3} \cdot \frac{\pi(2n-1)}{4(k+1)} + \frac{1}{3} \cdot \frac{\pi(2m-1)}{4k}$, $m, n \in \mathbb{Z}^+$. *We shift the value of m and n to find the minimum change of the central angle of the reflection point. The minimum change* $\frac{\pi}{12k}$ *is the minimum interval of two reflection points. As the central angle for the reflector on ring k is* $\frac{\pi}{2k}$, *there are at most six reflection points at each layer of the reflector board. This completes the proof.*

We obtain the expressions of the central angle for each reflection point in ring $k+1$ following the same procedure of ring k. We examine the distribution of reflection points on each reflector in ring $k+1$ based on simulation results, and found that at each layer from the ring 5 to the ring 50, there are average ten reflection points on the board of the ring $k+1$. One hole may be reused by a large number of reflection points for different reflection paths, i.e., the distance between two reflection points is small enough to overlap with each other. With the reuse ratio equal to the ratio of reused points to the total number of reflection points, Fig. 5 shows that the reuse ratio is high and increases when the ring number becomes larger.

3.3 Configuration for hot-spot traffic

Since the above techniques enable a large number of server-to-server wireless links, Diamond can implement a network-wide reconfigurable topology for balancing the identified hot-spot traffic, which contributes to high throughput and effective routing.

Configuration problem: The wireless configuration is determined by the network controller in DCNs. The controller input is a traffic demand matrix where an entry describes the traffic demand between a pair of servers. Given a hybrid topology G, we can construct its *interference graph* G_I[1] to describe the conflict relations among all the wireless links based on offline measurements [18]. The objective of our wireless configuration is to select the optimal *independent set* (IS) [2] to minimize the maximum link utilization of the entire network during each scheduling period. We thus have an integer linear programming (ILP) problem HLBP (Hybrid Load Balance Problem). Our HLBP problem mainly differs from previous study Firefly [19] on the additional wireless interference constraint on IS selections. Finding all the ISs is NP-complete in general [14]. We find that even the state-of-art ILP solver LINGO or ILP toolbox in MATLAB may take above tens of minutes to solve the HLBP.

[1] The *independent graph* G_I of G is defined as a graph where each link in G corresponds to a vertex in G_I, and if two links have conflict in G, then there is a link between them in G_I.

[2] An *independent set* (IS) in an interference graph G_I is defined as a vertex subset where any two vertices do not conflict.

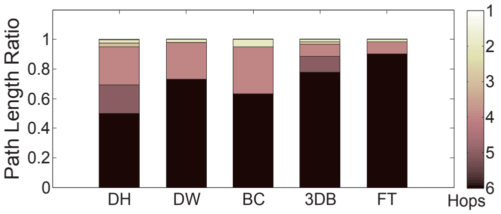


Figure 6: Path length ratio of different topologies

There are some existing studies [21–23, 34–36] on finding an approximate IS solution in some special interference graphs. In the following, in order to make Diamond support various types of antennas, we turn to the development of a fast heuristic solution for a general interference graph.

Greedy scheduling: We design a greedy algorithm HDF (Highest Demand First) to provide a faster and simpler solution for HLBP. The algorithm assigns a weight value to wireless links related to the flows, and then selects a set of non-conflict wireless links that maximize the sum of weights. We define the weight of a wireless link as the ratio between the flow demand and the link length. For a link with reflections, the link length is the total geometric length of the reflection path. The intuition is that a link can provide larger benefit when serving higher flow demand over a shorter link length, as a shorter wireless link allows for smaller interference range and higher SNR thus higher link capacity. We greedily select the links with the largest weight to build first and remove the links that conflict with the selected links. Next, the traffic demands are split into their shortest paths. Denote the minimum remaining capacity of links along a path as the *path capacity*. The server pair with the highest demand first splits the traffic to transmit over the set of shortest paths in proportional to the path capacity. Then the remaining link capacities are updated and the procedure repeats until no server pair is left. The gap between HDF and the optimal solution is evaluated in §7.

3.4 Random networking for high capacity and low delay

Since the wireless resources are rich in Diamond, after offloading the hot-spot traffic by HDF, some wireless radios may be left unused, particularly when the number of hot spots in the network is not big in a scheduling period. Random networking is shown to have the features of small average path length, high path diversity and high server-to-server network capacity [31, 32]. Thus we expect that the random formulation of wireless links helps to shorten the path length in Diamond.

To verify this effect, we compare the percentage of the path length for all the server pairs under different DCN topologies with 512 servers in Fig. 6. DH is for Diamond where wireless links are built randomly; DW is for Diamond with the wired connections only; BC is the BCube topology [16]; FT is the Fat-tree topology [3]; 3DB is a Fat-tree topology augmented by 3D-beamforming radios at ToR switches [38]. We can see that the number of long paths in DW is larger than that in BCube. However, when introducing random wireless links, the ratio of short paths in DH is higher than all the other topologies. The short path length generally implies small hop delay and high end-to-end throughput due to fewer congestion points at intermediate routing hops [31, 32].

To benefit from the random networking in Diamond, we extend the IS selected by the HDF algorithm to a *maximal* IS, named the MIS, by randomly adding additional wireless links into the IS without creating conflict until no such kind of wireless link is available. The random formulation of wireless links in Diamond avoids the problem of complex wiring and costly management appearing in the previous work on using random wired links in DCNs [32].

4 Routing Design

Diamond is built upon a topology-adaptive network, while existing routing protocols often impose a relatively long convergence time when the topology changes [6]. For more efficient routing, we propose to use a set of strategies in Diamond.

4.1 Overall scheme

The setup of wireless links is performed by the Diamond controller periodically. We denote the time interval for the controller to execute the wireless reconfiguration as one *period*. At the beginning of each period, a set of operations will be performed as follows: (1) The controller computes the wireless configuration and the routing paths for hot-spot server pairs using the methods described in §3, and sends out the instructions to both servers and their associated ToR switches. (2) The servers receiving the configuration instructions will adjust their antenna directions accordingly.

To summarize, there are three choices for a server or ToR switch to route its traffic. First, a server or ToR switch tries to match the routing rules designated by the controller. If matched, it delivers the packet accordingly. This first choice helps to balance the hot-spot traffic following the controller's decisions. Otherwise, it opportunistically utilizes its available wireless radios (if it is a server) or the available radios on its rack (if it is a ToR switch) to create a short-cut hop to the destination. This second choice contributes to shorter routing path through opportunistic hybrid routing. If no wireless radios are proper to use, it delivers the packet to the next-hop node following a default wired routing path. This last choice efficiently bounds the worst-case performance by routing through the wired herringbone.

4.2 Default wired routing

For the Diamond topology introduced in §2.3, a 3-tuple (x, y, z) labels a server at the xth row, yth column and zth layer. For simplicity, we use a 3-tuple $(x, y, 0)$ to label a ToR switch on the xth row and yth column. Fig. 7 shows a simple example to route from an arbitrary source server $s_1 = (x_1, y_1, z_1)$ to a destination server $s_2 = (x_2, y_2, z_2)$. Let $w_1 = (x_1, y_1, 0)$ and $w_2 = (x_2, y_2, 0)$ denote their corresponding ToR switches respectively. The shortest wired routing path can be established as follows. First, the packet routes from s_1 to w_1 and then we change one of the two coordinates of source ToR switch w_1 at a time to match that of switch w_2: $(x_1, y_1, 0) \rightarrow (x_2, y_1, 0) \rightarrow (x_2, y_2, 0)$. Finally, the packet routes from w_2 to s_2. Note that each coordinate change corresponds to hops through a virtual switch.

Suppose we apply de-Bruijn structure to implement the virtual switch, and the Diamond topology has totally $H = 2p$ layers and N rings. Then we need $4p$ ports per ToR switch, where $2p$ ports connect to the servers on the rack and $2p$ ports are used for constructing the de-Bruijn on its row and column. Since the diameter of a de-Brujin graph is $\log_p N$, the path length through a virtual switch (i.e., the path length between two ToR switches on one row or column) can be bounded by $\log_p N$. Based on the above routing procedure, we have the property:

Property 3. *The network diameter, which is the longest shortest path among all the server pairs, of Diamond is bounded by* $2\log_p N + 2$.

Since the Diamond with H layers and N rings can support totally $n = 2(N^2 + N)H$ servers, we have the diameter of Diamond as $O(\log_p n)$. Compared to conventional approaches (e.g., the Fat-tree [3] or VL2 [15] topology) which has a constant diameter but the number of switch ports increase with the number of servers, Diamond has much better scalability. As the server number increases, its network diameter extends logarithmically while the port number can be kept as a constant. This is similar to the recursion-based DCN topology such as BCube [16] and DCell [17]), which also has a logarithmic diameter when keeping a constant number of switch ports.

4.3 Opportunistic hybrid routing

The wired herringbone of Diamond provides the basic assurance of the connectivity and route length bound. Now we integrate the wireless transmissions into the default wired paths opportunistically. Suppose a server $s_1 = (x_1, y_1, z_1)$ receives a packet for a server $s_2 = (x_2, y_2, z_2)$, and the ToR switches of s_1 and s_2 are $w_1 = (x_1, y_1, 0)$ and $w_2 = (x_2, y_2, 0)$. The server s_1 is equipped with two radios, which are pointed to servers $s_1' = (x_3, y_3, z_3)$ and $s_1'' = (x_4, y_4, z_4)$, respectively. Define a **hamming distance** $\mathscr{D}(s_1, s_2)$ as the number of the unmatched coordinates between the tuples s_1 and s_2. Then the value range of $\mathscr{D}(s_1, s_2)$ is $\{0, 1, 2, 3\}$.

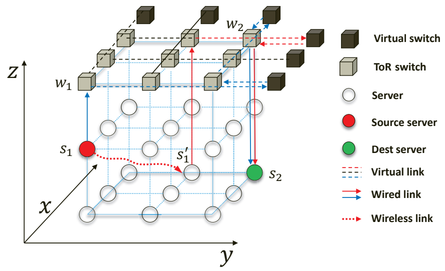


Figure 7: Opportunistic hybrid routing in Diamond

To perform the *opportunistic hybrid routing* (OHR), each server in Diamond simply follows two steps for the packet forwarding: (1) Call $d_1 = \mathscr{D}(s_1, s_2)$. If all three are matched (i.e., $d_1 = 0$), then it is the destination server. (2) Call $d_1' = \mathscr{D}(s_1', s_2)$ and $d_1'' = \mathscr{D}(s_1'', s_2)$. If $d_1' < d_1$ or $d_1'' < d_1$, forward the packet to the server s_1' or s_1'' accordingly through a wireless radio. Otherwise, forward the packet to the switch w_1 by default.

Similar to the servers, each ToR switch in Diamond forwards the packet as follows: (1) Call $d_1 = \mathscr{D}(w_1, s_2)$. If the first two are matched (i.e., $d_1 = 1$), it forwards the packet to s_2 directly; Otherwise, it randomly chooses one coordinate among the unmatched ones. Assume that ToR switch picks x_1 where $x_1 \neq x_2$, then the default next hop is $w_f = (x_2, y_1, 0)$. (2) For each server s_i in the rack, suppose its wireless radios point to two servers s_i' and s_i''. Call $d_i' = \mathscr{D}(s_i', s_2)$ and $d_i'' = \mathscr{D}(s_i'', s_2)$. If $d_1' < d_1$ or $d_1'' < d_1$, forward the packet to the sever s_i in the rack; Otherwise, forward the packet to the ToR switch w_f by default.

4.4 Fault-tolerance

The redundancy of available paths between any pair of servers make Diamond attractive for fault-tolerance. There are two types of failures to handle in Diamond: node failure and link failure. Three classes of node failure should be considered: (a) switch failure, (b) server failure and (c) wireless radio failure. A link failure will be resulted from a node failure, or the change of the environment such as the blocking of wireless communications due to the human movement in the RRSs. Clearly, due to the nested structure of Diamond, any single node or link failure does not lead to the network disconnection. We describe link failures here because other node failures trigger the same responses.

In Diamond, each server has three different output links to forward the packets: (a) forward to the ToR switch it connects to; (b) forward to one of its two wireless radios. When a server finds one of its output links fails, it removes that wired/wireless connection from its connection list, and chooses one of the remaining available output links as its next hop based on the routing rules described in Section 4.3. Benefited from the distributed routing property of OHR, the routing paths can be recovered quickly in Diamond to ensure high fault tolerance.

5 Discussion on deployment issues

Circle vs. Polygon reflector. We have so far suggested using the flat mental board as the reflector to facilitate its economic production and easy deployment. If the cost is not a concern, however, a curved mental reflector would allow the wireless communication range of each server to be larger than that of the flat reflector for the same constraint on reflection times. Considering that the curved reflectors are used to construct the circumcircle of each ring, with the ring number varying from 5 to 100 and the reflection times set to be within three, we find that the average wireless communication range per server in the polygon case is above 80% that of the circle case. When the number of rings is smaller than 5, both cases ensure the communications of all the servers of the entire ring. The results indicate that using flat reflector is a better choice for the deployment in a large-size data center.

Design of virtual switch. Diamond introduces a virtual switch to interconnect the ToR switches on a line (row or column) and the virtual switch can be implemented by any existing interconnection structures, e.g., the tree-based structure [3, 15] or cube-based structure [2, 16], with different trade-offs between the cost and performance. However, the number of ports required by a virtual switch on different rows and columns may not be the same in Diamond. Consider a Diamond topology with $2n$ rows and $2n$ columns, the port numbers of virtual switch from row r_1 to r_n are $\{2,4,6,...,n-2,n\}$. The uneven port numbers make it difficult to deploy conventional interconnection structures as some structures do not scale continuously [3,16,17]. To address this issue, we suggest using one virtual switch to interconnect two rows (or two columns) together to make a balance of the port number. Then each virtual switch requires $n+2$ ports by combining every two rows as (r_1,r_n), (r_2,r_{n-1}), (r_3,r_{n-2}) and so on. We can obtain the same result as that of $2n+1$ rows and columns by excluding the median row and column.

Rack density and wireless link number. To provide an idea of the deployment density of Diamond, we give an example. A room of data center with the size $100\times100m^2$ can hold 2k racks if using Diamond, and hold 3.7k racks if using the conventional row-based architecture, so the density of the conventional architecture is about 1.8 times that of Diamond. The lower rack density in Diamond ensures a proper space for both the wireless transmissions and cooling when the network scales up. However, our simulation results with different room sizes of a data center show that, the server-to-server throughput in Diamond on average doubles that of a conventional three-layer fat-tree DCN topology for the same room size [3]. As Fig. 4 shows, if we limit the reflection times of a path to be less than two, for a medium-size data center with 1000 servers, there will be more than 0.1 million potential wireless links available for use. The rich wireless links contribute a lot to the network-wide adaptive topology formulation and can support efficient routing and fault-tolerance in Diamond.

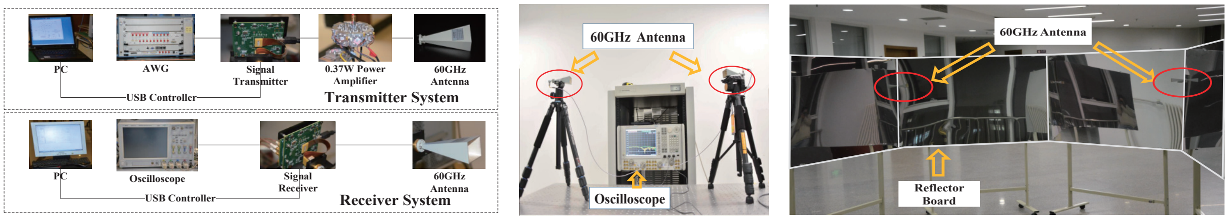
Table 1: Total cost of different DCN architectures

Topology #	Cost (k$)					Power (kw)
	NIC	Switch	Radio	Wire	Total	
FatTree	80	2080	-	80	2240	3486
3DB	80	2080	192	80	2432	3486
FireFly	80	416	2400	16	2912	4281
Diamond	240	832	1920	32	3024	3428

Cabling complexity. The cabling complexity is an important issue to consider in the deployment of DCNs. Despite their contributions to big performance improvements, both the tree-like topologies [3] and recursion-based topologies [16, 17] introduce complex cabling among racks and thus high maintenance cost in practice. This is because the physical row-by-row rack deployment does not work well with their logical tree or recursive topologies. In contrast, the cabling in Diamond is much easier with its wired structure simplified to be several rows and columns both logically and physically. As Fig. 2(a) shows, the row lines and column lines are independent from each other and thus are simple for both cabling and maintenance.

Cooling and maintenance. Heat dissipation is important for a data center to run healthily. In conventional DCN architectures, the most challenging heat issue comes from the closely placed racks in multiple rows. Since the rack density in Diamond is both lower and more balanced (i.e., the distance between any two neighboring racks is similar) than conventional architectures, the heat is distributed more evenly and lightly. For better cooling effect in Diamond, we suggest piping the cooling air from bottom to top in each ring. In addition, we suggest leaving four gaps at the polygon corners evenly on each ring to form four tunnels through the innermost to the outermost, through which the engineers can go inside each ring for device maintenance. When there is human moving inside, some wireless links may be blocked and failed. However, Diamond can handle the failure of wireless links easily (§4.4) and fast redirect the flows to wired links until the wireless link is available again.

Moreover, the antenna steering delay may be an issue to affect the system performance. The delay of steering 60GHz antenna can potentially be controlled within 250us if using phase array technology [18], while if deploying FSO in Diamond, the steering delay can be within 0.5ms using Galvo mirrors [19]. To further alleviate



(a) Transmit control panel (b) Direct communication (c) Multiple reflection

Figure 8: 60GHz antenna testbed for Diamond

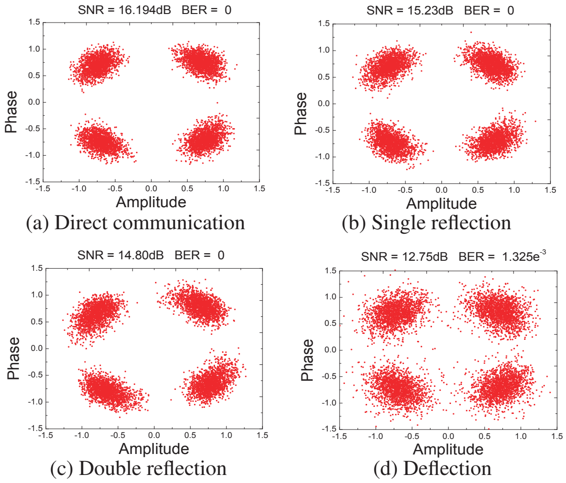
the side effect, our system ensures that the transmissions through wireless links during the antenna steering to be easily migrated to the stable wired links.

Deployment cost. A set of hybrid DCNs are proposed recently, such as the 3D-beamforming(3DB) [38] (8 radios per rack) and FireFly [19]. We use Fat-tree to represent the conventional wired architecture and compare the cost of different architectures in Table. 1. We consider the cost and power of NICs on the server, switches, wireless radios and wires. We conservatively estimate each wireless radio costs $60 [39], each 40-port switch costs $1040, each port in the NIC costs $5 and needs 5W [16], each port in the FSO device costs $150 [19], and an average cost of $1 per meter for cabling [19] and $1 per square meter of absorbing paper. We assume the reflectors used in Firefly, 3DB and Diamond have negligible cost. All the architectures hold 16 thousand servers. We can see that although Diamond uses a large number of radios, its cost is only 24% higher than that of 3DB because it uses 60% fewer switches. This trade-off is reasonable as a larger number of wireless links are enabled in Diamond than 3DB. Firefly can offer higher bandwidth at a higher deployment cost. However, the ceiling mirror it requires may not be applicable in most modern data centers. An alternative solution is to replace 60Ghz radios in Diamond with FSO devices, which will provide similar performance as Firefly without the need of deploying ceiling mirrors but at a higher deployment cost.

6 Implementation

We implement a 60GHz testbed to evaluate the transmission performance of our architecture under different wireless communication conditions.

Experiment setup: To demonstrate the feasibility of 60GHz wireless communication in our architecture, we build a testbed (Fig. 8a) to carry out the relevant experiments. The testbed was composed by Vubiq Networks Inc's commercial millimeter wave transceiver components, self-designed 60GHz Power Amplifier and AINFO Inc's 60GHz rectangular waveguide horn antenna. The system enables 60 GHz experiments on the use of integrated transmitter/receiver waveguide modules. 60GHz Power Amplifier is placed at the end of the transmitter to increase the transmission power. It has a gain of 30dB and a saturated output power of 0.37W. The testbed encodes the data file with LPDC and applies the QPSK modulation to generate the waveform. The receiver module samples the signal and recovers the original data file.



(a) Direct communication (b) Single reflection

(c) Double reflection (d) Deflection

Figure 9: Measured constellation diagram: performance of different transmission ways

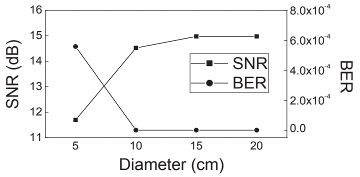
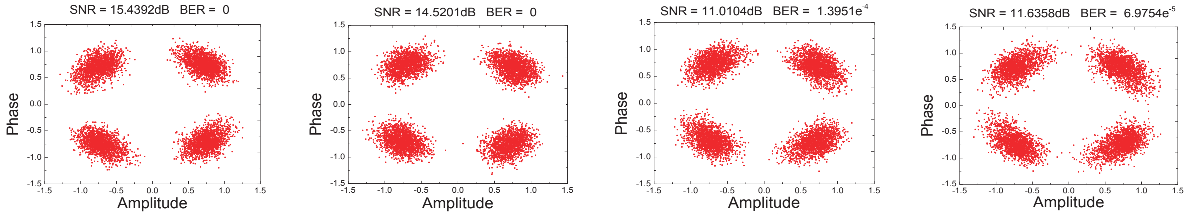


Figure 10: Performance over different hole sizes

We first carry out four experiments, including the direct communication, communication through single reflection, communication through double reflections and communication through deflection (i.e., the misalignment of two communicating antennas). In this group of experiments, to ensure the transmission ability of the architecture, the distance between the sender radio and the receiver radio is set to 25 m. The communication rate is 2.5 Gbps and the LPDC encoding rate is 3/4. We show the results in Fig. 9. For the second group of experiments, we change the hole size to test the performance of precise reflection for both the single and double reflection cases. To make an accurate measurement of hole size, the distance between the sender and receiver is set to 3m. The results are presented in Fig. 10 and Fig. 11.

Experiment result on signal reflection: As Fig. 9



(a) Single reflection without absorbing

(b) Single reflection on one 10cm x10cm hole

(c) Double reflections without absorbing

(d) Double reflections on two 10cm x10cm holes

Figure 11: Measured constellation diagram: performance of precise reflection

shows, the direct communication and the communications through single reflection and double reflections present a good communication quality and the corresponding SNR are 16.194 dB, 15.23 dB and 14.80 dB respectively. For all the experiments on our testbed, the measured data rates of both the directional and reflectional 60GHz links are shown to keep a value above 2.5Gbps over a distance of 25m. Therefore, the bandwidth of 60GHz wireless link is high enough for multiple-gigabit data transmissions in Diamond.

Experiment result on receiver alignment: During the measurement, we find that the communication quality through deflection changes with the deflection angle between two radios. As Fig. 9d shows, when the deviation angle becomes 20°, the SNR is 12.75 dB, which is the critical value of the communication quality. When the deviation angle further increases, the communication quality becomes too bad for the receiver to decode the original data. This indicates that our 60GHz radio is highly directional and has a small main-lobe width less than 20°, which contributes to a small angular interference to other radios when constructing the wireless interference graph. At the same time, the main-lobe angle provides a certain degree of fault tolerance on the antenna alignment between two servers in Diamond. We studied the impact of antenna misalignment through simulation with the above experimental parameters as input, and our result show that the average flow throughout drop is within 10% when the misaligned degree is within ±20°, which demonstrates that Diamond has a good tolerance to the fault as a result of the misalignment of antennas.

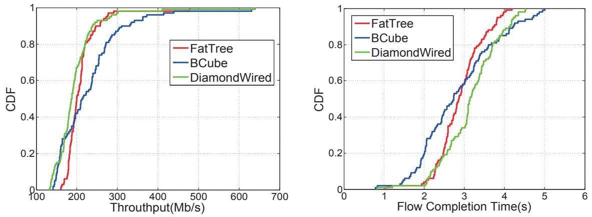
Experiment result on the precise reflection: We examine the impact of hole size on the reflector and show the single-reflection performance in Fig.10. We are not showing the results with hole size larger than 20cm, because they are the same as the 20cm case. We can see that when the hole size is 10cm, the SNR gets a slight decrease but BER is kept at zero. When the hole size further decreases to 5cm, the SNR drops quickly and results in the transmission failure.

After obtaining the proper hole size as 10cm, we measure the constellation diagram for both the single and double reflections. Fig. 11(a) and Fig. 11(c) show the results of reflections without any absorbing materials on the reflector. Fig. 11(b) and Fig. 11(d) show the corresponding results with one 10cm x10cm hole on each reflector. We can see that the transmission performance keeps nearly the same for both cases. Another interesting finding is that for double reflections, the SNR even gets slightly better when the reflectors are full of absorbing paper with only one hole left. The gain may be achieved as a result of the reduction of the multiple-path interference with the use of absorbing material. This demonstrates the feasibility of using precise reflection in Diamond.

7 Simulation

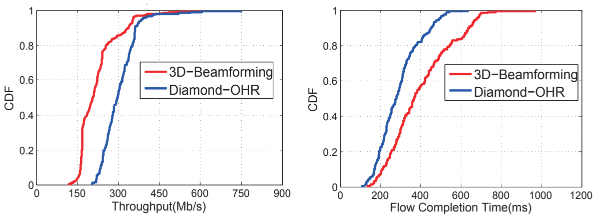
Setup and workloads. Our simulations are performed by a customized flow-level simulator. We use the same settings of TCP for the flow-level simulator as that utilized in [4], where the additive increase factor of flow rate is set to 15 MB/s. The wireless transmission follows the general physical interference and path loss model [10]. The related wireless parameters, such as the signal fading due to the misalignment of antennas, are all set following the testbed-based measurement results shown in Section 6.

For comparative analysis, we consider two classes of typical DCN topologies respectively: (1) wired topology and (2) hybrid topology. In the first part, we evaluate the performance of the wired backbone of Diamond (named Diamond-Wired) and other typical wired DCN topologies. The wired link capacity is set to 1Gbps, and we use Fat-tree [3] and BCube [16] as the representatives for the tree-based DCN topology and the recursion-based DCN topology respectively. In the second part, we evaluate the performance of Diamond and the state-of-art hybrid architecture 3D-beamforming [38]. We apply Fat-tree as the oversubscribed core for 3D-beamforming. Since 3D-beamforming deploys the wireless radios only at the ToR layer, to make a fair comparison, we apply two radios on top of each rack for both 3D-beamforming and Diamond. Thus, only the first layer of servers in Diamond are equipped with wireless radios and the radio numbers are the same for both topologies. To compare the performance only under distributed routing, we further disable the HDF (Highest Demand First algorithm) function and only use the OHR (Opportunistic Hybrid



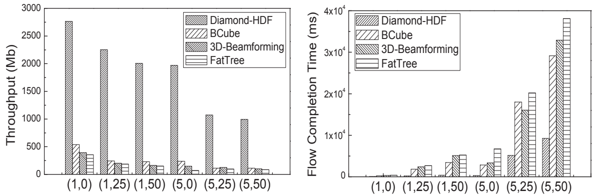
(a) Flow Throughput (b) Flow Completion Time

Figure 12: Flow Performance of Wired Architectures



(a) Throughput (Long Flows) (b) Completion Time (Short Flows)

Figure 13: Flow Performance of Hybrid Architectures



(a) Flow Throughput (b) Flow Completion Time

Figure 14: Flow Performance for Traffic Patterns

Routing) routing in Diamond (named Diamond-OHR), while 3D-beamforming uses ECMP routing. Limited by the memory space of our simulator, the number of rings in Diamond is set to six.

To compare the performance on load balancing and fault tolerance, we evaluate Diamond and other DCN topologies under different traffic patterns and number of node failures. The HDF routing and wireless radios are all enabled for Diamond (named Diamond-HDF) in the comparison cases. We transfer 200 random flows with their sizes set within 200MB, and show the performance results of flow throughput and flow completion time.

Performance of wired architecture. In Fig. 12a, we can see that BCube performs the best while Diamond-Wired has similar flow throughput as Fat-tree. The number of flows whose throughput is larger than 300Mbps takes 10% in BCube, while the percentage is less than 1% in the other two topologies. This is because that DiamondWired simplifies its wired backbone by using much fewer switches and wires. Similar trends on the performance of flow completion time can be found in Fig. 12b.

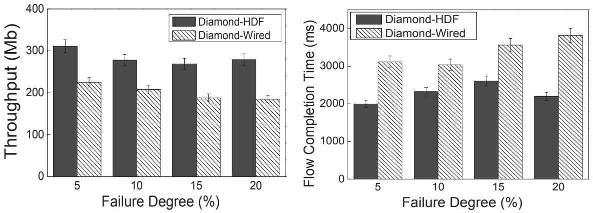
Performance of hybrid architecture. Consider the original traffic as long flows. We add another 200 random short flows (whose average size is one tenth that of the original traffic) to study the performance of mixed flows in hybrid architectures. In Fig. 13a, the throughput of long flows in Diamond-OHR is higher than that of 3D-beamforming. The number of long flows whose throughput is larger than 225Mbps takes above 90% in Diamond, while the number takes less than 40% in 3D-beamforming. Moreover, in Fig. 13b, the maximum completion time of short flows in Diamond is about 25% less than that of 3D-beamforming. In Diamond, a larger number of concurrent wireless links can be supported to increase the transmission capacity, which contributes to both higher throughput for long flows and smaller completion time for short flows.

Performance of load balancing. Following the prior work [19], we use a uniform model where flows between pairs of racks arrive independently with a Poisson arrival-rate as the baseline. We also consider the hotspot model [23], where in addition to the uniform baseline, a subset of rack pairs have higher arrival rates and larger flow sizes. We use a tuple (X,Y) to describe the hotspot traffic model: the X element represents the average flow size, where 1 denotes the average flow size is 100MB, and 5 corresponds to 500MB; the Y element denotes the percentage of the number of hot nodes.

As Fig. 14 shows, the flow performance of the four topologies deteriorates as expected when increasing the average flow size and the number of hot nodes. Diamond-HDF performs the best, providing the largest flow throughput and lowest flow completion time. Benefited from the rich server-level wireless links, the throughput of Diamond is about 5 times that of other topologies in the lightest traffic case $(1,0)$, and 9 times that of the other topologies in the worst traffic case $(5,50)$. Correspondingly, the flow completion time of Diamond is about 70% lower than that of other topologies. This demonstrates the high performance gains of Diamond-HDF and its capability of effectively balancing the load upon heavy traffic.

Performance of fault tolerance. In Fig. 15, we evaluate the flow performance of Diamond-HDF and Diamond-Wired when different percentages of nodes fail. To ensure that every flow can be routed under the node failures, we first randomly disable certain percentage of nodes and then randomly generate 100 flows to transmit for the remaining nodes. As Fig. 15 shows, the flow throughput of both the Diamond-HDF and Diamond-Wired decreases with the increasing node failure ratio. However, the flow throughout of Diamond-HDF decreases much slower than that of Diamond-Wired. Considering the failure ratio from 0% to 20%, the flow throughput of Diamond-HDF decreases about 13% while Diamond-Wired decreases about 28%. This illustrates the graceful performance degradation of Diamond-HDF for node failures. Similar trends on flow completion time can be found in Fig. 15b.

Performance of wireless reconfiguration. In Table 2, we compare the computation delay and performance of the greedy solution HDF in Diamond with the optimal



(a) Flow Throughput (b) Flow Completion Time

Figure 15: Flow Performance for Fault Tolerance

solution (named Full-ILP) of the HLBP problem. We use the ILP solver LINGO to compute the global optimal solution of ILP for routing (we obtained the same results when using the ILP toolbox in MATLAB for calculation). Limited by the memory constraint of LINGO, we evaluate the scales of Diamond with up to 5 rings which contains totally 60 racks and each rack holds 48 servers. We can see that the computation delay of Full-ILP increases quickly with the number of rings while HDF keeps a stable and low computation delay around 30ms. The tradeoff is HDF gets up to 15% gap on the performance of throughput and flow completion time when compared with Full-ILP. For a practical network scale within 20 rings, Full-ILP can not provide the solution in reasonable time, while HDF still achieves a low delay within 100ms, which is comparable to the feasible scheduling overhead illustrated in [4].

8 Related Work

Conventional data center: There exist prevalent hot spots in hierarchical data centers [3, 15, 18, 39], which limits the DCN performance. Many DCN architectures have been proposed to address the hot-spot problem in tree-based data center networks. Some efforts [30–32] propose to construct a random networking topology to achieve smaller network diameter, less hot spots and higher performance than state-of-art structured architectures. But the wiring and routing are quite challenging in a totally random wired network. In [16, 17], authors propose to build the network recursively to efficiently eliminate the structured bottleneck. However, the routing is restricted to follow its recursive structure, which does not consider the high dynamics in traffic demands and thus may lead to more hot spots.

Hybrid data center networking: Recent efforts turn to hybrid data center networking with flexible new networking components (e.g., the optical circuit switches, 60GHz wireless radios or FSO transceivers) to address the dynamic traffic demands [13,18,19,24,25,28,33,38]. Flyway first illustrates the feasibility of applying 60GHz wireless technology in DCNs [18]. The work in [38] further enhances the Flyway performance by using the ceiling reflector to bounce signals to avoid blocking on the 2D plane. Using the same method, Firefly explores the feasibility of running free-space-optical (FSO) transmissions in DCNs [19]. This method, however, requires a height-restricted ceiling and also complete clearance above racks, which is infeasible in most data centers due to the existence of air conditioning pipes and steel structures above the racks [1]. Moreover, existing methods only considered the local performance improvement at the rack level and part of network layers. In contrast, Diamond can run a larger number of network-wide wireless links (either 60GHz or FSO) without involving any engineering efforts to change the room plan above racks. Both wireless technologies can be applied in Diamond at the server level with different trade-offs: commodity 60GHz antenna is much cheaper and smaller than FSO transceivers while FSO has little interference footprint and longer transmission distance. With the decreasing cost of optical transceivers, FSO shows great promise to run in Diamond in the future.

Table 2: Performance of reconfiguration

Ring	Delay (ms)		Throughput	Flow Completion Time
#	Full-ILP	HDF	Gap	Gap
2	219	15	0.08	0.11
3	313	31	0.08	0.15
4	625	31	0.12	0.01
5	11625	32	0.15	0.15

9 Conclusion

We propose Diamond, a novel hybrid network architecture, to enable high capacity and seamless data transmissions over both wired and wireless network links. Specifically, we introduce the concept of Ring Reflection Space (RRS) to enable the wide deployment of wireless radios at servers and high number of concurrent wireless transmissions through low-cost multi-reflection over the metal, and develop a precise reflection scheme to reduce the wireless interference inside an RRS. The rich wireless resources allow Diamond to flexibly configure the network topology and form the transmission path to avoid creating hot traffic spots while enabling transmissions over random network topology for low delay. We also prove the scalability of the proposed architecture. We implement the proposed techniques over 60Ghz testbed and demonstrate its functionality. Our results from extensive simulations show that the cohesive structure of Diamond enables fine-grained and network-wide load balancing, effective routing and graceful fault-tolerance.

Acknowledgments

This work is supported by National Natural Science Foundation of China (no. 61120106008, 61422206), Tsinghua National Laboratory for Information Science and Technology (TNList). Xin Wang's research is supported by NSF CNS 1526843, and Xiangyang Li's research is supported by NSF ECCS-1247944. We would like to thank our shepherd Alex C. Snoeren and the anonymous reviewers for their valuable feedback and suggestions.

References

[1] Google data center image. `http://www.google.com/about/datacenters/gallery/#/all`.

[2] Abu-Libdeh, H., Costa, P., Rowstron, A., O'Shea, G., and Donnelly, A. Symbiotic routing in future data centers. In *SIGCOMM* (2011).

[3] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *SIGCOMM* (2008).

[4] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic flow scheduling for data center networks. In *NSDI* (2010).

[5] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center tcp (dctcp). In *SIGCOMM* (2011).

[6] Basu, A., and Riecke, J. Stability issues in ospf routing. In *SIGCOMM* (2001).

[7] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *IMC* (2010).

[8] Benson, T., Anand, A., Akella, A., and Zhang, M. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review 40*, 1 (2010), 92–99.

[9] Chen, K., Singla, A., Singh, A., Ramachandran, K., Xu, L., Zhang, Y., Wen, X., and Chen, Y. Osa: an optical switching architecture for data center networks with unprecedented flexibility. In *NSDI* (2012).

[10] Cui, Y., Wang, H., Cheng, X., Li, D., and Ylä-Jääski, A. Dynamic scheduling for wireless data center networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 24*, 12 (2013), 2365–2374.

[11] Curtis, A. R., Mogul, J. C., Tourrilhes, J., Yalagandula, P., Sharma, P., and Banerjee, S. Devoflow: scaling flow management for high-performance networks. In *SIGCOMM* (2011).

[12] de Bruijn, N. G., and Erdos, P. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen 49*, 49 (1946), 758–764.

[13] Farrington, N., Porter, G., Radhakrishnan, S., Bazzaz, H. H., Subramanya, V., Fainman, Y., Papen, G., and Vahdat, A. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *SIGCOMM* (2011).

[14] Garey, M. R., and Johnson, D. S. Computers and intractability: A guide to the theory of np-completeness. *WH Freeman & Co., San Francisco* (1979), 61–62.

[15] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. Vl2: a scalable and flexible data center network. In *SIGCOMM* (2009).

[16] Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., and Lu, S. Bcube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM* (2009).

[17] Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y., and Lu, S. Dcell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM* (2008).

[18] Halperin, D., Kandula, S., Padhye, J., Bahl, P., and Wetherall, D. Augmenting data center networks with multi-gigabit wireless links. In *SIGCOMM* (2011).

[19] Hamedazimi, N., Qazi, Z., Gupta, H., Sekar, V., Das, S. R., Longtin, J. P., Shah, H., and Tanwer, A. Firefly: a reconfigurable wireless data center fabric using free-space optics. In *SIGCOMM* (2014).

[20] Kandula, S., Sengupta, S., Greenberg, A., Patel, P., and Chaiken, R. The nature of data center traffic: measurements & analysis. In *SIGCOMM* (2009).

[21] Li, X.-Y. Multicast capacity of wireless ad hoc networks. *IEEE/ACM Transactions on Networking (TON) 17*, 3 (2009), 950–961.

[22] Li, X.-Y., Tang, S.-J., and Frieder, O. Multicast capacity for large scale wireless ad hoc networks. In *MOBICOM* (2007).

[23] Li, X.-Y., and Wang, Y. Simple approximation algorithms and ptass for various problems in wireless ad hoc networks. *Journal of Parallel and Distributed Computing 66*, 4 (2006), 515–530.

[24] Liu, H., Lu, F., Forencich, A., Kapoor, R., Tewari, M., Voelker, G. M., Papen, G., Snoeren, A. C., and Porter, G. Circuit switching under the radar with reactor. In *NSDI* (2014).

[25] Liu, Y. J., Gao, P. X., Wong, B., and Keshav, S. Quartz: a new design element for low-latency dcns. In *SIGCOMM* (2014).

[26] 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*, 2 (2008), 69–74.

[27] Perry, J., Ousterhout, A., Balakrishnan, H., Shah, D., and Fugal, H. Fastpass: A centralized zero-queue datacenter network. In *SIGCOMM* (2014).

[28] Porter, G., Strong, R., Farrington, N., Forencich, A., Chen-Sun, P., Rosing, T., Fainman, Y., Papen, G., and Vahdat, A. Integrating microsecond circuit switching into the data center. In *SIGCOMM* (2013).

[29] Shin, J.-Y., Sirer, E. G., Weatherspoon, H., and Kirovski, D. On the feasibility of completely wirelesss datacenters. *IEEE/ACM Transactions on Networking (TON) 21*, 5 (2013), 1666–1679.

[30] Shin, J.-Y., Wong, B., and Sirer, E. G. Small-world datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), ACM, p. 2.

[31] Singla, A., Godfrey, P. B., and Kolla, A. High throughput data center topology design. In *NSDI* (2014).

[32] Singla, A., Hong, C.-Y., Popa, L., and Godfrey, P. B. Jellyfish: Networking data centers randomly. In *NSDI* (2012).

[33] Wang, G., Andersen, D. G., Kaminsky, M., Papagiannaki, K., Ng, T., Kozuch, M., and Ryan, M. c-through: Part-time optics in data centers. In *SIGCOMM* (2010).

[34] Wang, W., Wang, Y., Li, X.-Y., Song, W.-Z., and Frieder, O. Efficient interference-aware tdma link scheduling for static wireless networks. In *MOBICOM* (2006).

[35] Wang, Y., Wang, W., Li, X.-Y., and Song, W.-Z. Interference-aware joint routing and tdma link scheduling for static wireless networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 19*, 12 (2008), 1709–1726.

[36] Xu, X., Li, X.-Y., Wan, P.-J., and Tang, S. Efficient scheduling for periodic aggregation queries in multihop sensor networks. *IEEE/ACM Transactions on Networking (TON) 20*, 3 (2012), 690–698.

[37] Yu, M., Greenberg, A. G., Maltz, D. A., Rexford, J., Yuan, L., Kandula, S., and Kim, C. Profiling network performance for multi-tier data center applications. In *NSDI* (2011).

[38] Zhou, X., Zhang, Z., Zhu, Y., Li, Y., Kumar, S., Vahdat, A., Zhao, B. Y., and Zheng, H. Mirror mirror on the ceiling: flexible wireless links for data centers. In *SIGCOMM* (2012).

[39] Zhu, Y., Zhou, X., Zhang, Z., Zhou, L., Vahdat, A., Zhao, B. Y., and Zheng, H. Cutting the cord: a robust wireless facilities network for data centers. In *MOBICOM* (2014).

Ripple II: Faster Communication through Physical Vibration

Nirupam Roy, Romit Roy Choudhury

University of Illinois at Urbana Champaign

Abstract

We envision physical vibration as a new modality of data communication. In NSDI 2015, our paper reported the feasibility of modulating the vibration of a smartphone's vibra-motor. When in physical contact with another smartphone, the accelerometer of the second phone was able to decode the vibrations at around 200 bits/s. This paper builds on our first prototype, but redesigns the entire radio stack to now achieve 30 kbps. The core redesign includes (1) a new OFDM-based physical layer that uses the *microphone* as a receiver (instead of the accelerometer), and (2) a MAC layer that detects collision at the transmitter and performs proactive symbol retransmissions. We also develop two example applications on top of the vibratory radio: (1) a finger ring that transmits vibratory passwords through the finger bone to enable touch based authentication, and (2) surface communication between devices placed on the same table. The overall system entails unique challenges and opportunities, including ambient sound cancellation, OFDM over vibrations, back-EMF based carrier sensing, predictive retransmissions, bone conduction, etc. We call our system *Ripple II* to suggest the continuity from the NSDI 2015 paper. We close the paper with a video demo that streams music as OFDM packets through vibrations and plays it in real time through the receiver's speaker.

1 Introduction

Motivation: Project Ripple [34] is an attempt to enable communication through physical vibrations. The core idea is to harness the vibration motor (present in all smartphones and wearable devices) as a transmitter, and a motion sensor (like an accelerometer) as a receiver. When two smartphones come in physical contact to each other, the transmitter phone can vibrate to transfer bits of information. Transmission is even possible through other solid channels, such as between devices placed on a tabletop, or a finger ring communicating to a smartphone through bone conduction. While the exact application remains an open question (especially in the presence of NFC-like technologies), areas such as Internet of Things (IoT), intra-body networks, wearable security, and mobile payments are calling for new forms for short range communication. Qualities of a vibratory radio, including zero RF radiation, contact-only authentication, mass-scale availability, and intuitive usability, may together fill an emerging business need. This project is motivated by this "bottom up" thinking and focuses on pushing forward the vibratory capabilities.

Prior Work: Of course, the fundamental idea of utilizing vibration as a communication modality dates back to acoustics – speakers modulate bits of information into air vibrations that are picked up by microphones. Air vibrations were later extended to water, enabling under water communication [7, 6, 11] and various applications, such as SONAR [41]. In recent years, vibration through solids has been of interest, motivated primarily by the need for proximal communication. Authors in [22, 38] used Morse-style communication at 5 bit/s to exchange security keys between two mobile phones in contact. Last year, *Ripple* [34] broke away from ON/OFF communication, and developed a viable radio through techniques such as multi-carrier amplitude modulation, vibration braking, and simultaneous transmission over the 3 axes of the accelerometer. A self-sound cancellation technique also prevented acoustic eavesdroppers from decoding the sounds of vibration, offering improved security over RF based approaches. As a first attempt to vibratory radio design, Ripple achieved data rates of ≈ 200 bits/s, but left various challenges and opportunities unaddressed. This paper presents a subsequent work – Ripple II – aimed at a far more mature radio stack and two example applications.

Technical Core: Ripple II's core redesign entails the following: (1) Replacing the accelerometer with the microphone as a receiver of vibrations. The key challenge pertains to separating vibrations from ambient sounds "picked up" by the microphone. While the availability of a second microphone offers the opportunity for sound cancellation, vibrations partly pollute the second microphone as well. Moreover, techniques such as active noise cancellation are inadequate since residual phase mismatches – often tolerable in human hearing applications [37] – seriously affect demodulation. We develop variants of adaptive filtering schemes, enhanced with an understanding of the interference conditions. (2) We also discover an opportunity that allows the vibra-motor to partially sense ambient sound interference, through a phenomenon called back-EMF in electronic circuits. The transmitter extrapolates from this partial information, using curve fitting techniques, and develops a proactive symbol retransmission scheme. The problem is new to the best of our knowledge – unlike existing wireless systems, here the transmitter is aware of the receiver's interference conditions and can adapt at the granularity of symbols. This opens both challenges and opportunities.

System and Apps: We engineer a completely functional prototype, which entails a full OFDM stack, coping with ADC saturation, synchronization, error coding, interleaving, etc. Towards real applications, we develop a (clunky) wearable finger ring and demonstrate the viability of transmitting vibratory signals through finger bones. While signals attenuate through human tissues and muscles, effective bit rates of 7.41 Kbps is still possible, adequate for applications like two-factor authentication (i.e., when the user unlocks the phone, the vibratory password decoded by the phone serves as a second channel of authentication). We also explore a second application where devices are placed on tabletops, allowing for one-to-many multicast communication (e.g., a presenter sharing slides with all members in the meeting). Lastly, we include a video demo on our project website [3] – the demo shows the transmitter streaming music through OFDM packets over vibrations and the receiver's speaker playing it in real time.

Platform and Evaluation: Our evaluation platform is composed of laptops, signal generators, vibra-motor chips, microphone chips, and home-grown circuits that interconnect them. In the basic scenario, the vibra-motor is attached to a short pencil to emulate a "stylus" like device, which then touches a microphone chip to transfer information. We generate various ambient sounds in the lab, including soft and loud music, people talking, machine hums, loud thuds and vibrations, and their combinations. Our PHY and MAC layer schemes are evaluated in these settings, against metrics such as SNR gain, bit error rate (BER), throughput, etc. At the application layer, we compute end to end data rate under modestly realistic settings, such as the human wearing the (vibra-motor embedded) ring and touching the microphone chip. We emulate wrist watches as well (2.23 Kbps), and perform an informal user study to understand if they feel the vibrations. We also explore achievable bit rates for tabletop communication, with devices placed at increasing distances on wooden surfaces.

Next Steps: There is much room for continued research and improvement. First, we have little understanding of PHY capacity and MAC layer optimality; intuitively, we believe that modeling the devices and the channel can yield reasonable performance bounds. Second, the sound cancellation techniques can perhaps benefit from deeper signal processing expertise – we have initiated collaboration towards this goal. Third, microphones and accelerometers may together present new opportunities that remain untapped in this paper. Fourth, while our prior paper mitigated attacks on vibratory sounds, visual attacks still remain a threat – a high speed camera, with line of sight to the device, may be able to decode vibrations. Finally, we need guidance on other possible use-cases and applications [5] of vibratory radios. Our ongoing work is focused on all these aspects.

In summary, the contributions of this paper are:

- *An OFDM based vibratory radio with microphones as the receiver.* The PHY layer uses variants of adaptive filtering to isolate vibrations from ambient sounds at the microphone; the MAC layer develops a transmitter side carrier sensing mechanism and uses it for proactive symbol retransmission.

- *A completely functional system borne out of significant engineering effort.* The effort includes hardware circuits on bread boards, to drivers for the vibra-motor, to bone conduction and real-time music streaming. Instantiation of the system in two applications: touch based authentication and surface communication.

The overall architecture of Ripple II is illustrated in Figure 1. The rest of the paper expands on the main modules (shaded in gray) and briefly touches upon the techniques borrowed from literature, and the engineering effort in building the prototype.

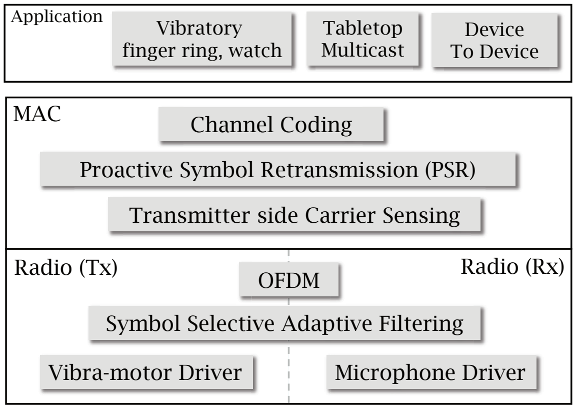

Application
Vibratory finger ring, watch
Tabletop Multicast
Device To Device
MAC
Channel Coding
Proactive Symbol Retransmission (PSR)
Transmitter side Carrier Sensing
Radio (Tx)
Radio (Rx)
OFDM
Symbol Selective Adaptive Filtering
Vibra-motor Driver
Microphone Driver

Figure 1: Ripple II's system architecture.

2 Development Platform and Overview

2.1 Vibratory Transmitter

A vibration motor (also called "vibra-motor") is an electro-mechanical device that moves a metallic mass rhythmically around a neutral position to generate vibrations. While there are various kinds of vibra-motors, a popular one is called *Linear Resonant Actuators (LRA)* shown in Figure 2. With LRA, vibration is generated by the linear movement of a magnetic mass suspended near a coil, called the “voice coil”. Upon applying AC current to the motor, the coil also behaves like a magnet (due to the generated electromagnetic field) and causes the mass to be attracted or repelled, depending on the direction of the current. This generates vibration at the same frequency as the input AC signal, while the amplitude of vibration is dictated by the signal’s peak-to-peak voltage. Thus LRAs offer control on both the magnitude and frequency of vibration. As an aside, most mobile phones today use LRA based vibra-motors.

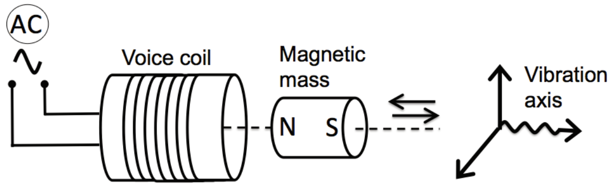


Figure 2: Basic sketch of an LRA vibra-motor.

We control the vibra-motor through an Agilent 33500B waveform generator, which is indirectly controlled by MATLAB running on a laptop. The laptop generates the desired digital samples; the waveform generator converts the samples to an analog wave and transmits to the vibra-motor. The peak-to-peak output voltage is stabilized at 5V, the maximum supported by the vibra-motor chip. We generate OFDM symbols through MATLAB and drive the motor as desired.

2.2 Microphone as a Receiver

Our prior work [34] used a vibra-motor as the transmitter and an accelerometer as the receiver[1]. The accelerometer demodulated vibratory QPSK symbols and corrected for errors using simple gray coding techniques. The low bandwidth of accelerometer chips (800Hz) proved to be the main bottleneck to link capacity, resulting in ≈ 200 bits/s. This paper breaks away from accelerometers and identifies the possibility of using microphones as a vibration receiver.

Like accelerometers, microphones also transduce physical motion to electrical signals using a diaphragm that responds to changes in (acoustic) air pressure. Figure 3 shows a microphone chip and the basic internal architecture – as the diaphragm vibrates inside a magnetic field, the produced electrical signals are amplified and sampled by an ADC. *Unsurprisingly, the diaphragm can also be made to vibrate by physically touching a vibra-motor to the microphone chip.* Since microphones are designed for greater sensitivity and operate over a wider frequency range, they can serve as a better receiver (an alternative to accelerometers). The tradeoff, however, is that the vibration measured at the ADC is actually an aggregate of the physical vibration and the air vibration from ambient sounds (e.g., people talking). Ripple II needs to isolate physical from acoustic vibrations to accomplish high bandwidth vibratory communication.

[1] Accelerometers are MEMS devices that transduce physical motion into electrical signals by measuring the extent to which a tiny seismic mass moves inside fixed electrodes (see [24] for details).

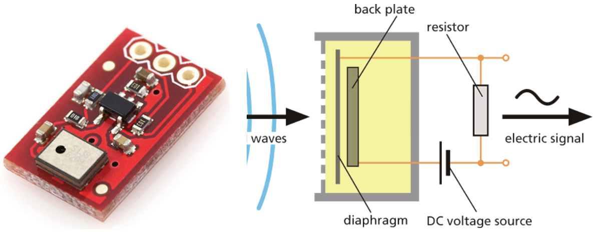


Figure 3: (a) MEMS microphone chip, the diaphragm hole near bottom left (b) Microphone circuit sketch.

Figure 4 shows our overall hardware set-up. The vibra-motor is taped to the back of a short pencil and the tip of the pencil now acts like a stylus, touching the microphone chip. Transmission bits produced by the laptop are converted to a signal waveform by the signal generator, which then drives the transmitter; the microphone decodes these bits through realtime processing on a laptop. The following subsections detail the technical modules in the PHY, MAC, and Application layers.

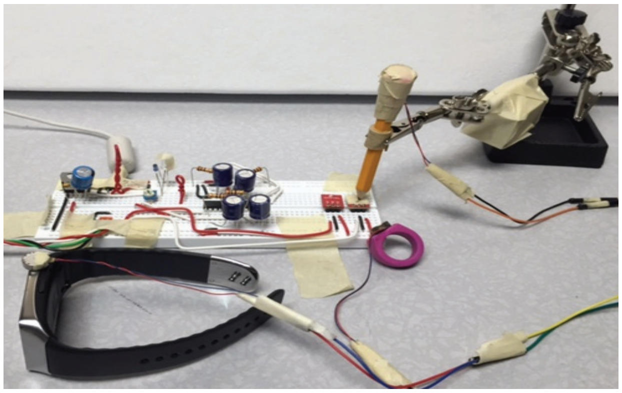


Figure 4: Ripple II’s experimentation set-up (3 vibra-motors attached to a pencil, ring, and watch). The stylus touching a microphone, the second microphone nearby.

3 PHY: Vibratory Radio

We begin with the design of the microphone-receiver, followed by our implementation of OFDM.

3.1 Separating Vibration from Sound

While the microphone offers larger bandwidth compared to the accelerometer, its sensitivity to ambient sound is a disadvantage. Unless filtered out, the vibration SINR will be low, especially in loud environments. We attempted various techniques (algorithms and hacks); we detail the ones that worked and touch upon the failures.

Covering the Sound Hole

The microphone chip has a circular opening (like a small hole) that exposes the diaphragm to air pressure. To prevent ambient sounds from polluting Ripple II's vibratory signals, we covered the hole with a stiff synthetic rubber sheet (somewhat like a stethoscope). However, when a vibrating object comes in contact with this rubber sheet, the air trapped inside the hole still oscillates, causing the diaphragm to produce the desired signals. Figure 5 compares the frequency responses of the altered and the standard microphones for vibration and sound, respectively. Figure 5(a) shows an average 18.2dB gain for vibration signals over the standard microphone; at some frequencies the difference is 43.8dB. On the other hand, Figure 5(b) shows that the average sound attenuation at the altered microphone is around 12.3dB. For both the signal (i.e., vibration) and the noise (i.e., ambient sounds), the higher frequency proves better (useful later in Section 5).

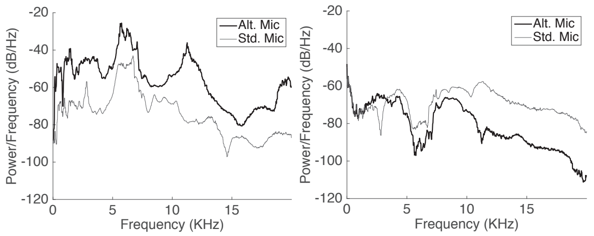


Figure 5: Covering the sound hole offers (a) improved vibration signal and (b) attenuated sound signals in comparison to the standard microphone.

Canceling Ambient Sound

Let us denote the vibration signal from the stylus as $V(t)$ and the ambient sound signal as $S(t)$. Ripple II aims to subtract $S(t)$ from the aggregate signal ($A(t) = V(t) + S(t)$)) received through the microphone. A second microphone present in many devices today is a natural opportunity. In an ideal case, the second microphone should only receive the ambient sound $S(t)$ and none of the physical vibration $V(t)$ since the stylus is not in direct contact with it. In reality, however, physical vibrations also leak into the second microphone. Also, both microphones are affected by a high intensity electrical noise, $E(t)$, caused by their common supply voltage. Frequencies of this noise range from 300Hz to 2500Hz and its amplitude is comparable to $V(t)$. Finally, the microphone output also includes a native hardware noise, typically assumed to be uncorrelated additive Gaussian, denoted N_1 and N_2 for the respective microphones.

Based on the above factors, the overall system can be modeled as shown in Figure 6. The signal output from the i^{th} microphone, Y_i can thus be expressed as:

$$Y_i = H_{v_i}V + H_{s_i}S + H_eE + N_i$$

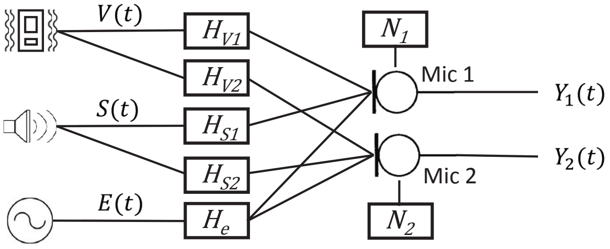


Figure 6: Modeling the signals and interferences at each of the microphones; H denotes the channel matrix and V, S, E, denotes vibration, sound, and electrical noise, respectively.

We note that extraneous physical vibrations may occur when Ripple II is transmitting information (for example, in a moving vehicle). Such vibrations are included in S since it is likely to affect both the microphones similarly. We also note that the electrical noise E is highly correlated and synchronized at both microphones, since they share a common power source. Under this model, our goal is to extract V from Y_1 and decode the content.

Failed Attempts (MIMO, NC, rPCA)

MIMO: We discovered early that electrical noise E can be removed effectively by low pass filtering Y_2 and subtracting from Y_1. Since E dominates and is phase matched across both microphones, the residue after subtraction minimally impacts V. Thus, we can rewrite $Y_i = H_{v_i}V + H_{s_i}S + N_i$. This appears to be in the form of MIMO and hence solvable without difficulty. Unfortunately, the channel matrix for ambient sound, H_{s_i}, cannot be easily measured since Ripple II has no control over the sound sources. Also, due to the time-varying nature, statistical estimates are difficult.

Classical Noise Cancellation seems applicable [25], however, the statistical nature of this algorithm does not mitigate phase mismatches. The result after subtraction does preserve the amplitude of the desired signal, which is often adequate for human perception [37]. In Ripple II, however, we need phase alignment too, or else, QAM based demodulation falters. Put differently, requirements to improve human hearing experience is less stringent than the requirements for data communication.

Robust PCA is an algorithm from 2009 used for background separation [8]. The technique builds on the result that, under certain conditions, a given matrix can be factorized into a sparse and low rank matrix. For instance, in a talk show video, static background walls could serve as the low rank matrix (due to high similarity across video frames) and the talking people could make up the sparse matrix. In our case, we envisioned the ambient sound to be sparse and the vibration to be low rank (since the cyclic prefix of OFDM symbols can be organized to look identical across time[2].). Unfortunately, we could not design the matrices to attain adequate amount of both sparsity and low rank-ness. During the short time shifts for which the OFDM vibration symbols were identical, the sound signal changed enough that they were not sparse. When sound proved to be sparse over longer time frames, the low rank-ness disappeared. The outcomes of factorization yielded marginal gain.

Symbol Selective Adaptive Noise Filtering

Adaptive filtering (AF) is an established technique that can accept the two microphones' signals as inputs, say ($Y_1 = V_1 + S_1$) and ($Y_2 = V_2 + S_2$), and can attempt to adapt the filter coefficients for Y_2 such that the $Y_1 - Y_2$ is V_1. Conceptually, AF bolsters Y_2 in the regions where it correlates well with S_1, and then subtracts from Y_1. This works best when S_1 and Y_2 are somewhat correlated to each other, but neither is correlated to V_1. However, in our system, when ambient interference is low (i.e., V dominates S), then Y_2 correlates well with V_1 – this is why AF subtracts away the vibratory signals from Y_1, defeating the purpose. However, we observe that if we could identify OFDM symbols that are in error (i.e., S dominates V), then perhaps *only the erroneous symbols* could be subjected to AF. Since S_1 and Y_2 would correlate well in such cases, the result of $Y_1 - Y_2$ could converge to V_1. Using this intuition, we design *Symbol Selective Adaptive Noise Filtering* (SANF), sketched in Figure 7.

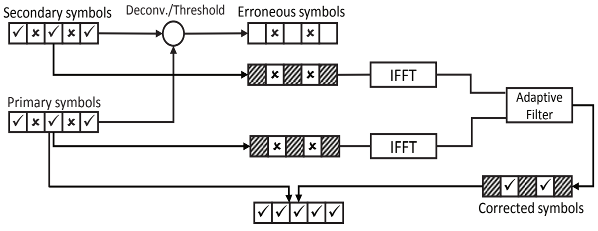


Figure 7: SANF infers erroneous symbols and only feeds these to the AF module.

Erroneous Symbol Detection. The main opportunity emerges from measurements that revealed that the vibratory channel responses at the primary and secondary microphones – H_{v_1} and H_{v_2} – maintain a constant ratio under light or no interference. This is likely due to the same solid channel between the two microphones. In the presence of sound, however, the same ratio gets polluted and thereby loses the constancy property (since sound varies over time). Thus, we first perform channel estimation for pilot subcarriers scattered across the OFDM symbol. We synchronize the secondary microphone and estimate the channel for that same pilot (the slight time offset does not affect due to the protection from the cyclic prefix). Now, deconvolution of the primary and secondary signal in the frequency domain yields the complex gain, α_p for each pilot p.

Recall, the goal is to estimate the pristine ratio of H_{v_1} and H_{v_2} in the presence of sound interference; the α_p we have is still polluted by sound interferences. Thus, we perform a least square estimation of the ratio and compute α^* for each subcarrier. Now, for any non-pilot symbol to be erroneous, the computed complex gain between the primary and secondary must be far from α^* for that subcarrier. Once the erroneous symbols are identified, we convert only those to the time domain, leaving the error-free subcarriers untouched. We obtain the time domain signals from both of the primary and secondary microphone and feed them to an adaptive filter for noise cancellation. The output of the adaptive filter is then demodulated to recover the vibratory symbols.

Amplifier Gain and Clipping

To maximize the power of the vibratory signal, we operate the receiver signal amplifier at near-maximum gain and leave just enough headroom for typical ambient sound (measured empirically). Of course, sometimes the ambient sound exceeds the headroom and drives the amplifier to *saturation* [33]. Figure 8(a) shows the output of the unsaturated amplifier; Figure 8(b) shows the saturated case – a truncated waveform. Unsurprisingly, this "clipping" effect spills energy into other frequencies, causing interference in an OFDM system. We alleviate such frequency distortion effects by replacing the flat saturation region with a cubic spline interpolation of the signal – Figure 8(c).

Our measurements also recorded consistent interference at lower frequencies ($<$ 500Hz), caused by a combination of winds from air vents, thermal noise from electrical equipment, as well as vibrations of the human hand while holding the transmitter. The vibra-motor also exhibits resonance frequency at around 232Hz, causing the system to destabilize due to the high power gain. We deemed it suitable to sidestep these problems and moved the transmission band to begin from 500Hz.

3.2 OFDM over Vibration

We implement OFDM [12] over the vibra-motor and microphone link. Although an engineering effort, we

[2] Without too much details, note that time domain signals can be shifted by several samples and yet, by OFDM design, they will map to the same frequency domain symbol – this is why we could generate low rank-ness.

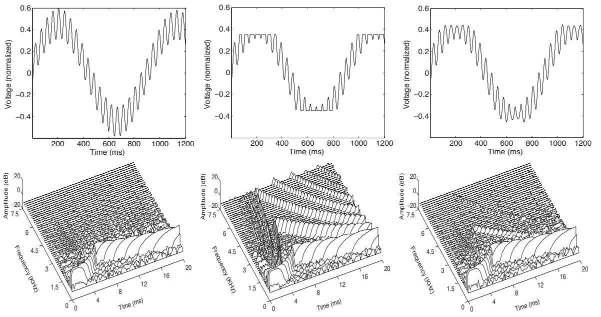


Figure 8: The waveform (1st row) and spectrogram (2nd row) of the (a) actual signal, (b) distorted signal after clipping, (3) corrected signal after spline interpolation.

briefly summarize the parameter selection process, particularly those influenced by the vibratory channel.

Channel Impulse Response

Although the vibratory channel is dominantly time-invariant and frequency selective, human factors such as hand movements and varying angle of contact inject variability. Measurements suggest similarity to a Rician fading model [40], with a strong line of sight path. The weaker multipath components are caused by the inertial movement of the motor mass – reverberation of the medium distorts the signal and multiple reflected/delayed replicas combine to create an elongated decaying response at the output. We measure the impulse response of our system using the *exponential sine-sweep* method [14] during which sinusoids of exponentially increasing frequency drives the motor. The output from the microphone is de-convolved with the weighted reverse sine-sweep to obtain the impulse response (the technique offers robustness against noise and non-linear distortions). Figure 9(a) and (b) show the measured impulse response and the corresponding *power delay profile* (PDP).

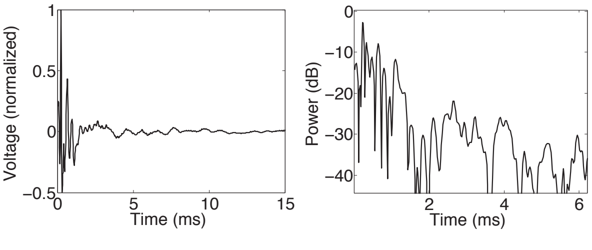


Figure 9: (a) Channel impulse response (b) Power delay profile of the vibratory channel.

Parameter Selection

Cyclic Prefix: The PDP shows 0.4ms before the multipath energy falls below 10dB of the highest peak, called "10DB maximum excess delay". This should be the separation between symbols to avoid inter symbol interference (ISI). We set the guard interval conservatively to 1ms, however, instead of leaving the channel idle during this interval, we insert 1ms of the last part of the symbol. This is called the cyclic prefix (CP) which helps cope with time synchronization errors without affecting the orthogonality of sub-carriers.

Subcarrier Bandwidth: The vibratory channel, as mentioned earlier, offers long channel coherence time, allowing for small subcarrier spacing. In practice, however, due to unpredictable phase noise, the inter carrier interference (ICI) becomes severe with small subcarrier spacing. On the other hand, the subcarriers become frequency selective for bandwidths larger than the coherence bandwidth of the channel. In such cases, the channel is no longer flat and hence equalization techniques falter [13, 35]. We measure the coherence bandwidth to be 480Hz (see Figure 10) – this is the width of the frequency-correlation function using a threshold of 0.95. We then choose the subcarrier bandwidth conservatively to 40Hz, less than the $\frac{1}{10}^{th}$ of the coherence bandwidth.

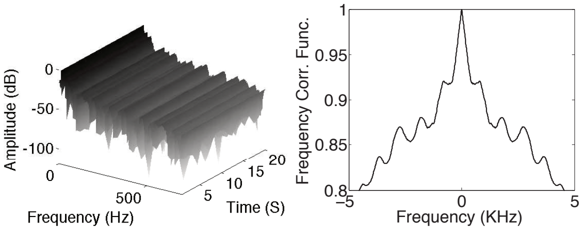


Figure 10: (a) Temporal stability of the channel, (b) The frequency-correlation function indicates the coherence bandwidth of 480Hz for the width threshold of 0.95.

Total Bandwidth: We choose the total bandwidth to be 12KHz, equal to the coherence bandwidth at correlation threshold of 0.7.

With this PHY layer in place, we now focus on a vibratory MAC layer, with the goal of reliably delivering packets to the receiver even under interference.

4 MAC Layer Design

Reliable packet delivery entails retransmitting a packet when it is received in error. In wireless systems, since the transmitter is unaware of the receiver's channel conditions, the error detection happens reactively, through an ACK from the receiver. Vibra-motors offer a new opportunity – we find that the receiver's interference conditions can be sensed at the transmitter through what is known as *back EMF*. Thus, the transmitter could potentially transmit and listen at the same time, infer symbol collisions, and retransmit symbols proactively. Efficiency can improve but some issues need mitigation.

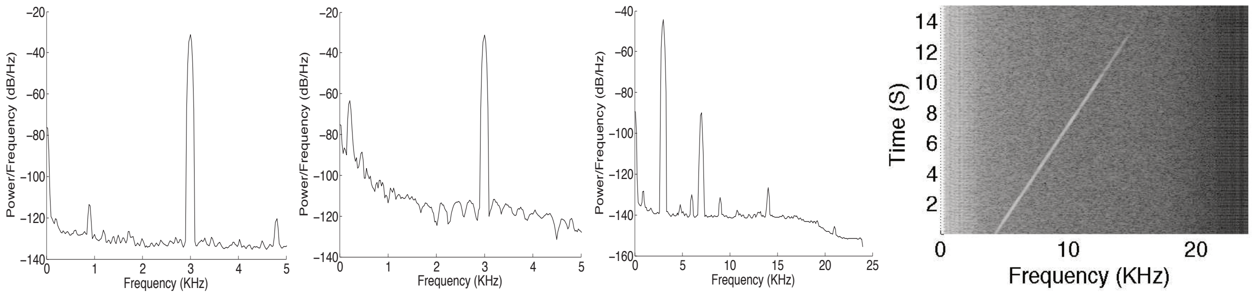


Figure 11: (a) Vibra-motor driven by a 3KHz voltage and no interference in the environment. (b) Interference introduced in the environment raises the noise floor, especially at lower frequency bands. (c) Clear detection of 7KHz interference caused by a nearby vibrator. (d) Spectrogram of acoustic chirp detected through back-EMF – the chirp was played through a speaker placed 4 feet away.

4.1 Sensing Interference from Back-EMF

Back-EMF is an electro-magnetic effect observed in magnet-based motors where relative motion occurs between the current carrying armature/coil and the magnetic field. In our vibra-motor, when the permanent magnet oscillates near the coil, the flux linkage with the coil changes due to the driving voltage and/or vibration noise. According to the Faraday's law of electromagnetic induction [16], this changing flux induces an electromotive force in the coil. Lenz's law [39] says this electromotive force acts in the reverse direction of the driving voltage, called *back-EMF of the motor*. As the rate of change of the magnetic flux is proportional to the speed of the magnetic mass, the back-EMF serves as an indicator of the extraneous vibration experienced by the mass.

Unsurprisingly, the interfering vibrations generate subtle movements of the vibra-motor mass, causing the voltage changes around a small resistor to be in milli-volts (below the ADC noise floor)[3]. We design a low noise amplifier, limiting the parasitic inductance/capacitance, to amplify this voltage 100*x* before feeding it to the ADC sampling circuit. Figures 11(a,b) show the difference between interference-free and interfered transmissions, as sensed through back-EMF. The noise floor increases, especially at lower frequencies where the interference is dominant. Figure 11(c) shows another case where a 7KHz interferer – a second interfering vibra-motor – is placed on the same table as our experiment; the transmitting vibra-motor detects the corresponding spike at 7KHz. We also played an acoustic chirp on a speaker 4 feet away from our devices – Figure 11(d) shows the chirp spectrogram, a reasonable reproduction of the actual. The findings extend hope that back-EMF can be useful to designing transmitter-side collision inference protocols.

4.2 Vibratory Interference

Before moving into protocol design, we characterize the nature of vibratory interference experienced by the microphone. Interferences are broadly of two kinds. (1) Ambient acoustic sounds, such as people talking, background music, machine hums, etc. and (2) physical vibrations caused by objects such as running table fans, taps and thuds on table-tops, and even natural vibration of human hands when they are holding the devices. Figure 12 shows the spectral graph of several example interferences, measured in isolation. The key observation is that interferences are heavily biased to the lower frequency bands; frequencies higher than 6KHz are rarely impacted.

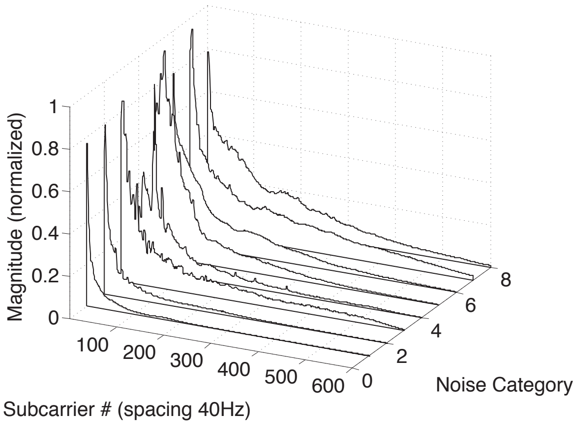


Figure 12: Spectral properties of various interferences occurring in the natural environment.

Figure 13 shows the 3D contour of acoustic interference across frequency and time – the interference stems from loud human voices. The key observation is that for any given frequency, the signal amplitude of the interference rises with time, reaches a peak, and decays again. This characteristic is highly common in a wide range of interferences, primarily because instantaneously starting or stopping strong signals is difficult. Occasionally, we find

[3] The measuring circuit samples the induced current as a voltage drop across a series resistor. We keep this resistor value below 0.02% of the motor's coil resistance so that the electrical property of the system remains unaffected.

certain machines capable of producing a sudden spike, however, their decay is still slow. We leverage back-EMF along with these properties of the interference to design a MAC protocol, called *Proactive Symbol Recovery* (PSR).

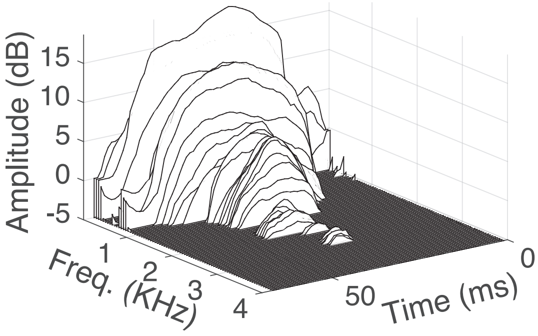


Figure 13: 3D contour of acoustic interference across frequency and time.

4.3 PSR Protocol: The Problem Definition

The protocol problem can be abstracted as follows. Consider a packet P composed of many OFDM symbols, $[S_1, S_2, S_3, ...]$, each symbol composed of n subcarriers $[f_1, f_2, f_3, ...f_n]$. Figure 14 shows the pictorial representation of such a packet, in the form of a time-frequency grid. Assume that the gray region denotes the incidence of interference, essentially the top view of Figure 13. Now, with back-EMF, the transmitter is able to sense receiver-side interference, however, the sensing is not accurate. To be able to reliably detect interference (i.e., reduce false positives), the transmitter can increase the sensing threshold – interference detected above this threshold is strongly indicative of actual interference. Assume that the interference above a given threshold is the black region in Figure 14.

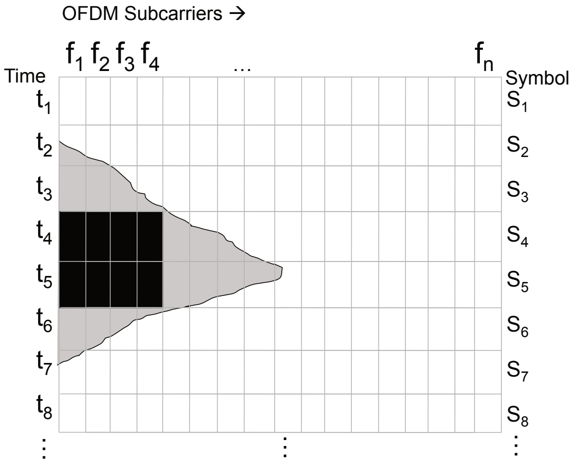


Figure 14: A packet represented in terms of OFDM symbol, each symbol to be transmitted over time.

The protocol question is: *which symbols should the transmitter retransmit, and when?* Transmitting only the symbols that are affected by the black color may still leave too many erroneous symbols – the coding scheme at the receiver may not be able to recover the packet. The transmitter essentially needs to estimate the symbols affected by the gray region too, and retransmit a subset of those symbols [23, 19]. Clearly, not all the gray-color affected symbols need to be transmitted since the coding scheme can indeed correct for some errors.

A second question pertains to interference adaptation. Once interference is detected at time t_4, the transmitter must adjust the subsequent transmissions to cope with the interference. Any adjustments – such as rate control – would need to be communicated to the receiver through some control information. However, unlike packets, symbols are not prefaced with headers; dedicating some subcarriers to a control channel will be wasteful in general. Under this constraint, the protocol needs to adapt to interference and concisely convey its adaptations to the receiver. The basic problem is new to the best of our knowledge, since existing protocols assume that the receiver has better estimates of error than the transmitter [23, 19]. In our case, the transmitter is better aware of the interference but has no control bits to convey its adaptations.

4.4 Proactive Symbol Recovery Protocol

The PSR protocol develops 2 heuristics – interference extrapolation and implicit control signaling – described next.

(1) Interference Extrapolation. Only the contour of the interference within the black region (in Figure 14) is visible to the transmitter – one could metaphorically envision it as the "part of the iceberg above water". Based on the visible shape, the transmitter may be able to extrapolate the "submerged" shape, generating an estimate of the gray region. Our measurements have consistently indicated that the interference decay is well-behaved, of course with some jitter. Hence, we model this as a curve fitting problem, and use a 3*rd* order cubic spline (the high frequency jitters are not captured). Given multiple silhouettes, one per-subcarrier, we pick the silhouette whose peak is at 80^{th} percentile among all peaks. Using this we develop an estimate of the gray region.

(2) Implicit Control Signaling. As mentioned earlier, the transmitter needs some control bits for signaling its actions to the receiver. To this end, we use a simple interleaving idea from the basics of signal processing. Specifically, when alternate subcarriers are loaded with data (and the ones in-between left empty), the time domain representation of the OFDM signal exhibits two identical copies (Figure 15). We call this the $2x$ interleaving mode. When every $4th$ subcarrier is loaded, the time domain signal shows 4 identical copies of the same signal. The receiver recognizes these identical copies in time domain and decodes the control information. In frequency domain, it extracts the data from every $2nd$ (or $4th$) subcarrier and ignores the others. Of course, we are aware that the control bits are not free – the $2x$ and $4x$ interleaving modes reduce the bandwidth. However, we also note that energy on the loaded subcarriers increases – a $2x$ mode exhibits a 3dB gain (nearly double), lowering chances of demodulation error.

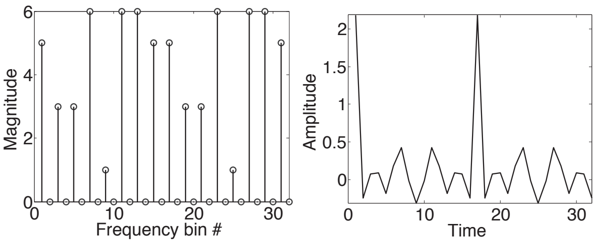


Figure 15: (a) 2x interleaving in frequency, (b) identical signal parts in time domain.

Protocol Design: We now describe the basic operation of the PSR protocol (we continue to refer to the toy example in Figure 14). When no interference is detected by the transmitter's back-EMF sensor (i.e., until time t_4), symbols are sent as usual. Upon detecting interference at t_4, the transmitter records the symbol that was affected (namely, S_4), and performs the subsequent symbol transmissions (S_5) at $2x$ interleaving mode. This continues until the interference has subsided below the transmitter's threshold. At this point, the transmitter performs the extrapolation using the interference decay data, starting from the last-observed interference peak. The interpolation suggests that the receiver may continue to experience interference until some time in the future, say till t_7. Therefore, the transmitter continues symbol transmissions in $2x$ mode, after which it falls back to no-interleaving. Observe that this interleaving mechanism is akin to halving the rate, except that it helps inform the receiver about the rate reduction.

Ideally, the interference extrapolation may help recover the symbols S_6 and S_7, however, symbols S_2 and S_3 could also be heavily interfered. To this end, the transmitter also extrapolates the front portion of the interference, and remembers the symbols that need retransmission. Once all the symbols have been transmitted, it now retransmits these symbols (S_2, S_3, and S_4 in this toy case), at the appropriate interleaving mode permissible by the then channel conditions. Importantly, the receiver must identify that these symbols are actually duplicates of prior symbols. Hence, the transmitter marks the start of these retransmissions with a $4x$ interleaved packet – the packet includes indices of all symbols that are being retransmitted. The encoding of indices is efficiently done to utilize the fewest bits possible, telling the receiver how many retransmissions to expect and which prior symbols to replace. The receiver demodulates all the symbols, performs the appropriate replacements, and feeds them through the decoder.

Coding for Error Correction: Needless to say, extrapolation will incur errors, and back-EMF sensing will experience false negatives. This will leave erroneous symbols at the receiver even after retransmissions. In fact, it would be inefficient for the transmitter to recover all symbols since the decoder at the receiver would be able to correct for some of them anyway. We implement a standard 2/3 convolutional code, with constrain length 7, to cope with inherent symbol errors in the transmission. We implement a hard decision Viterbi decoder with trace back depth of 30 to recover the bits. To cope with heavy bursts in error, we use an interleaver to spread out the bursts.

5 System Evaluation

5.1 Complete Hardware Prototype

Figure 16 shows the complete interconnection of the hardware elements in Ripple II. Very briefly, the receiver (on the left side) draws power from the USB port of a Dell laptop (or any mobile device or raspberry-pi/arduino) serving as the controller. Instead of using a separate ADC, we abuse the *Line-in audio input port* of the laptop, which comes equipped with a high speed ADC and a driver to push samples to user space. We connect signals from each microphone to one of the channels in the line-in port with the help of a three-conductor (TRS) audio jack. We run the appropriate driver to sample the signal at 48KHz, 16bit stereo mode.

The transmitter (shown on the right side) also uses a similar approach. The software controller generates digital samples that are converted to analog via the DAC of the audio port. This output signal (with appropriate amplification and shaping) feeds into the vibra-motor, which is in turn attached to the stylus or ring. We sample this line-in port at 48KHz to collect the back-emf signal along with the reference voltage. Offline processing is performed in MATLAB; realtime music streaming is performed on GNURadio.

5.2 Performance Results

We present end to end results first, followed by zoomed in results from acoustic noise cancellation (SANF) and proactive symbol retransmission (PSR). Our final results

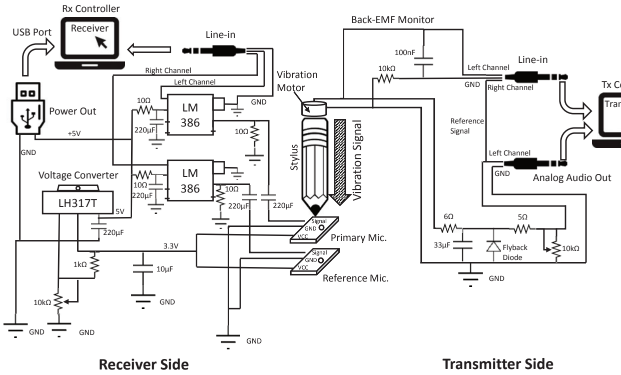


Figure 16: The complete hardware internals of Ripple II.

are drawn from 100+ sessions of experiments, each session either $1-3$ min. long, and entails vibratory transmission against diverse ambient sounds, ambient vibrations, modulations, etc. We collected 800 samples of ambient sound (e.g., supermarket ambience, in classroom noise, music nearby, etc.) and 15 ambient vibrations (e.g., walking, moving in a car, tapping on table). Half of sessions were against the natural lab sound conditions; for the other half, we played external ambience sounds through a speaker and generated vibrational noise through an external vibra-motor placed on the table. As a baseline we use the basic OFDM microphone receiver running on our hardware platform (including the covered sound hole). We compare this baseline against (1) baseline + coding, (2) baseline + coding + SANF, and (3) baseline + coding + SANF + PSR.

Ripple II Results:
Figure 17(a) shows the CDF of throughput gain computed from all the experimentation data, across all possible noise environments. The communication link operates in a high bit error rate (BER) regime and coding schemes perform worse than expected. The median gain with SANF is around 10%, with a small fraction of cases leading to negative gain. However, PSR brings appreciable benefits, mainly from retransmitting erroneous symbols and bringing the errors below the tolerable threshold. Median throughput gain with PSR is 26.6%. Figure 17(b) reports the breakup of raw throughput under various ambient sound categories. Under mechanical sound spikes alone, the performances of SANF and PSR are weak – the interpolation in PSR falters, while SANF's symbol error detection scheme is not sensitive enough. However, in other categories of noises, throughput improves – the median throughput in the "All" noise category is ≈ 27 Kbps.

SANF Results:
Figure 18(a) zooms into symbol selective aspect of SANF, and shows the fraction of symbols corrected over normal adaptive noise filtering. The correction gain improves with higher SNR, but falls beyond 15dB. This is because at > 15dB SNR, SANF is unable to detect the symbol errors correctly since the interference is less pronounced – the inability to identify the erroneous symbols derails adaptive noise filtering. The sensitivity curve captures this behavior, suggesting that the symbol correction efficacy is both a function of SNR and sensitivity. Figure 18(b) shows the gain across each subcarrier – the graph is for the best SNR, 15dB.

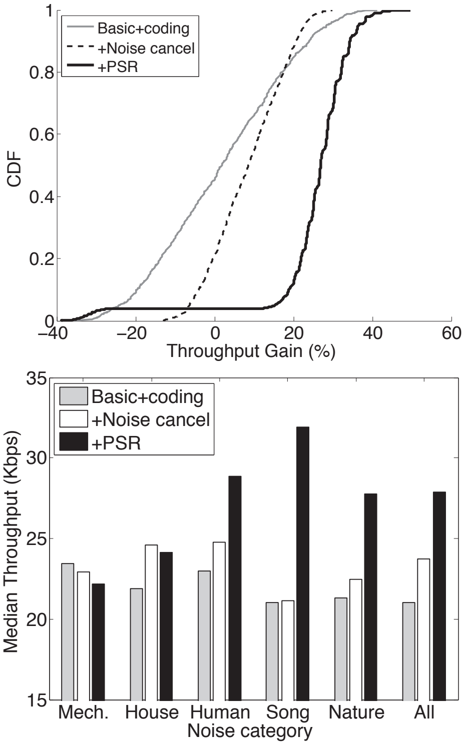


Figure 17: (a) Throughput gain across all experiments. (b) Median throughput across different ambient sound categories.

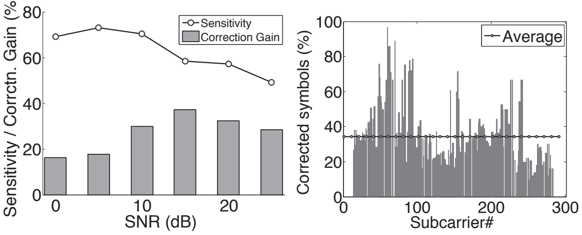


Figure 18: (a) Variation of SANF's cancellation gain and sensitivity against increasing SNR; sensitivity is the fraction of erroneous symbols detected by SANF. (b) The noise cancellation gain as the percentage of erroneous symbol per subcarrier.

PSR Results:
The core design elements in PSR pertains to (1) back-EMF based sensing and extrapolation of the interference,

and (2) reducing symbol errors via $2x/4x$ interleaving (expected to increase energy). To evaluate extrapolation, we first identify the set of truly erroneous symbols that should have been retransmitted by the transmitter. We know the set of symbols that PSR actually retransmitted. From these two sets, we compute the precision and recall of PSR, reflecting the combined efficacy of back-EMF sensing and interpolation. Figure 19(a) shows the results – the precision is strong but the recall is weak, indicating that PSR is conservative. This is expected/desirable since we intend to not retransmit excessively, which reduces inflation of the packet and also allows the decoder to correct for the residual errors. Of course, there is room to tune the interpolation scheme and the back-EMF sensitivity – we leave this to future work.

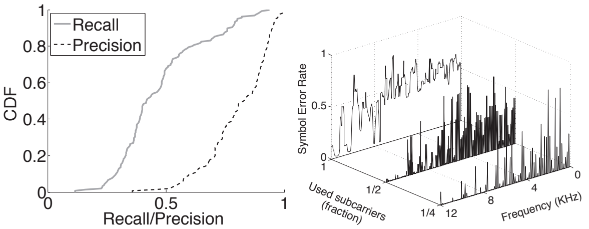


Figure 19: (a) Precision and recall to evaluate the back-EMF sensing and interference extrapolation scheme. (b) The per subcarrier symbol error rates using all, 1/2, and 1/4 of the subcarriers, while the noise power is constant.

Figure 19(b) shows the reduction in symbol error rate when half and one-forth subcarriers are loaded with data (recall we denoted this as $2x$ and $4x$ modes of transmission). Under heavy channel interference, $2x$ mode substantially reduces symbol errors, offering effects similar to rate control. However, the $2x$ mode also implicitly includes a control bit that the receiver can recognize. Measurements show that the control signaling was near perfect, meaning the receiver almost always extracted the correct data from $2x$ and $4x$ transmissions.

5.3 Applications and Capabilities

We explore potential applications of Ripple II, namely a vibratory ring and watch; tabletop communication; and device to device transfers.

(1) Finger Ring for Authentication

We envision touch based two-factor authentication – a user wearing a Ripple II ring or watch could touch the smartphone screen and the vibratory password can be conducted through the bones. The core notion generalizes to other scenarios, including unlocking car doors, door knobs, etc. While a usable system would need maturity in interfaces, energy, etc., this section only discusses the communication aspects of through-bone transmission. Figure 20(a) shows the crude finger ring prototype, placed on the index finger of the user. For our prototype, the ring is powered by a battery located outside the ring and connected via long wires. The cylindrical vibra-motor is placed horizontally on the finger to maximize area of contact, however, placement influences communication.

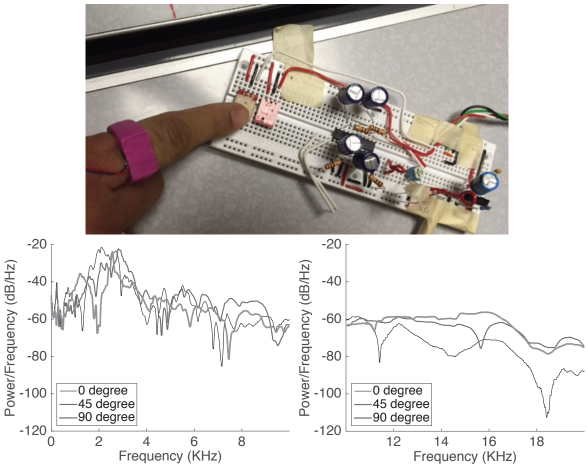


Figure 20: (a) Finger ring operated at 8KHz. (b) Incidence angles affect lower frequencies less. (c) Higher frequencies in a piston oscillator become directional and hence delivers less energy in unaligned directions.

Figure 20(b,c) shows the variation of signal power for 3 different incidence angles between the vibra-motor and the finger – incidence angle defined as the angle between the finger bone and the direction along which the vibrator mass oscillates (which is perpendicular to the base of the cylinder). Evidently, at lower frequencies, the incidence angle does not impact the signal, however, at higher frequencies the higher incidence angles reduce SNR. Moreover, higher frequencies are also less effective for signal propagation through the human body. Thus, we decide to operate the ring at 90° incidence but focus the power budget to within 8KHz.

We also performed similar experiments with a watch – pasting the vibra-motor on the wrist-bone below the watch. Performance degrades as expected, due to a longer conduction path from the wrist to the microphone. The table below summarizes results. 5 student volunteers experimented with our prototype and none of them were able to feel or hear the vibrations at all.

	Bandwidth	Modu.	Code	Tput:Kbps
Ring	8 KHz	QPSK	1/2	7.41
Watch	3 KHz	QPSK	1/2	2.23

5.4 Tabletop Communication

Multicast communication is often useful – a group picture at a restaurant needs to be shared with everyone in the group; presentation slides need to be shared in a meeting. We envision placing all phones on the table, near each other, and performing one vibratory multicast. Figure 21 shows the outcome of such an experiment – we used the stylus to touch different locations on a table, while 2 microphone receivers were at fixed locations on this otherwise empty table. Even at nearly 2 feet away, the throughput is around 4Kbps (the X-axis has duplicate values since there were multiple distinct locations at the same distance from the microphone).

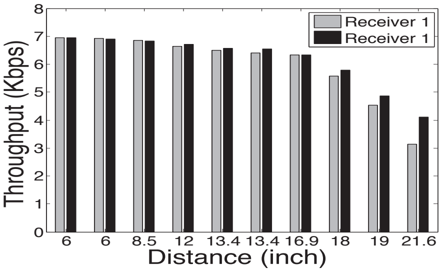


Figure 21: Throughput against varying tabletop range.

5.5 P2P Money Transfer

In developing regions, mobile payments may be viable with basic phones with vibra-motor and microphones. Perhaps a USB stick can transfer data to phones/tablets on physical contact. Such apps obviously need higher data rates and some may require real time operation. The table below shows possibilities when vibra-motor on stylus' and smartphones are touched to microphones. We have also built a demonstration of a real time music streaming system over vibrations – please see video demo here: `http://synrg.csl.illinois.edu/ripple/`

	Bandwidth	Modu.	Code	Tput:Kbps
Stylus	12 KHz	16 QAM	2/3	29.19
Phone	12 KHz	16 QAM	2/3	26.13

6 Related Work

Vibratory communication: Authors in [38] and [22] were the first to conceive the idea of communicating through physical vibrations. They both encode vibrations through (ON/OFF) Morse code, with pulse durations of around one second (i.e., 1 bits/s). This is adequate for applications like secure pairing between two smart phones, or sending a tiny URL over tens of seconds. Our prior work in NSDI 2015 [34] developed a fuller vibratory radio through multi-frequency modulation, self-jamming based security, and resonance braking, ultimately translating to 200 bits/s. Ripple II is a push-forward of the Ripple project, but with microphone as the receiver, and augmented with a new PHY/MAC layer offering 150*x* throughput gain. Ripple II still preserves Ripple's security properties via self-sound cancellation. Dhwani [31] and Chirp [2] address conceptually similar problems, although on the acoustic platform; vibra-motors bring about new set of challenges and opportunities. Technologies like Bump [1, 28, 36, 9, 20, 18, 26] use accelerometer/vibrator-motor responses to facilitate secure pairing between devices. TagTile [4] uses high frequency sound to achieve association between phones and point-of-sale devices. However, these techniques are primarily designed for few bits of exchange; Ripple II aims high bitrate transmission with the same ease as Bump and Tagtile. Further, as indicated by researchers [38, 17], the lack of the dynamic secret message in Bump-like techniques makes them less secure in the wild. These modes also require Internet connectivity and trusted third party servers to function, none of which is needed in Ripple II.

Vibration generation and sensing: Creative research in the domain of haptic feedback has investigated the state-of-the-art in electro-mechanical vibrations [32, 10]. Applications in assisted learning, touch-augmented environments, and haptic learning have used vibrations for communication to humans [30, 15, 21, 32, 10]. However, the push for high communication data rates between vibrators and microphones/accelerometers is unexplored to the best of our knowledge. Off late, personal/environment sensing on mobile devices has gained research attention. Applications like (sp)iPhone [27] and TapPrints[29] demonstrate the ability to infer keystrokes through background motion sensing. While many more efforts are around activity recognition from vibration signatures, this paper aims to modulate vibration for communication.

7 Conclusion

Ripple II is an attempt to enable touch-based vibratory communication between a vibra-motor and a microphone. We develop a vibratory radio at the PHY and MAC layer, and explore a few possible applications in authentication, device to device streaming, and table-top communication. While additional work is needed to attain maturity, we believe this paper is a concrete step towards demonstrating an alternative communication mode, that has remained relatively unexplored in the past.

Acknowledgement

We sincerely thank our shepherd Dr. Jonathan Smith and the anonymous reviewers for their valuable feedback. We are also thankful to Prof. Pramod Viswanath and Shaileshh Venkatakrishnan for the various discussions on this topic. We are grateful to Qualcomm, Huawei, and NSF (grant CNS-1430033) for partially funding this research.

References

[1] Bump Technologies. `http://blog.bu.mp`. Accessed: 7th February, 2016.

[2] Chirp. `http://www.chirp.io`. Accessed: 7th February, 2016.

[3] Ripple webpage. `http://synrg.csl.illinois.edu/ripple/`. Accessed: 7th February, 2016.

[4] Tagtile report. `http://www.mybanktracker.com/news/2011/10/07/tagtile-easier-reward/`. Accessed: 24th September, 2015.

[5] ADKINS, J., FLASPOHLER, G., AND DUTTA, P. Ving: Bootstrapping the Desktop Area Network with a Vibratory Ping. *Ann Arbor 1001* (2015), 48109.

[6] AKYILDIZ, I. F., POMPILI, D., AND MELODIA, T. Underwater acoustic sensor networks: research challenges. *Ad hoc networks 3*, 3 (2005), 257–279.

[7] BURDIC, W. S. *Underwater acoustic system analysis*. Prentice Hall, 1991.

[8] CANDÈS, E. J., LI, X., MA, Y., AND WRIGHT, J. Robust principal component analysis? *Journal of the ACM (JACM) 58*, 3 (2011), 11.

[9] CASTELLUCCIA, C., AND MUTAF, P. Shake them up!: a movement-based pairing protocol for cpu-constrained devices. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services* (2005), ACM, pp. 51–64.

[10] CHO, Y.-J., YAND, T., AND KWON, D.-S. A New Miniature Smart Actuator based on Piezoelectric material and Solenoid for Mobile Devices. In *The 5th International Conference on the Advanced Mechatronics, ICAM* (2010), pp. 615–620.

[11] COATES, R. F. *Underwater acoustic systems*. Halsted Press, 1989.

[12] DEBBAH, M. Short introduction to OFDM. *White Paper, Mobile Communications Group, Institut Eurecom* (2004).

[13] ENGELS, M., AND PETRÉ, F. *Broadband fixed wireless access: a system perspective*. Springer Science & Business Media, 2006.

[14] FARINA, A. Simultaneous measurement of impulse response and distortion with a swept-sine technique. In *Audio Engineering Society Convention 108* (2000), Audio Engineering Society.

[15] FEYGIN, D., KEEHNER, M., AND TENDICK, F. Haptic guidance: Experimental evaluation of a haptic training method for a perceptual motor skill. In *Haptic Interfaces for Virtual Environment and Teleoperator Systems, 2002. HAPTICS 2002. Proceedings. 10th Symposium on* (2002), IEEE, pp. 40–47.

[16] GALILI, I., KAPLAN, D., AND LEHAVI, Y. Teaching Faradays law of electromagnetic induction in an introductory physics course. *American journal of physics 74*, 4 (2006), 337–343.

[17] HALEVI, T., AND SAXENA, N. On pairing constrained wireless devices based on secrecy of auxiliary channels: The case of acoustic eavesdropping. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 97–108.

[18] HALPERIN, D., HEYDT-BENJAMIN, T. S., RANSFORD, B., CLARK, S. S., DEFEND, B., MORGAN, W., FU, K., KOHNO, T., AND MAISEL, W. H. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (2008), IEEE, pp. 129–142.

[19] HAN, B., SCHULMAN, A., GRINGOLI, F., SPRING, N., BHATTACHARJEE, B., NAVA, L., JI, L., LEE, S., AND MILLER, R. R. Maranello: Practical Partial Packet Recovery for 802.11. In *NSDI* (2010), pp. 205–218.

[20] HOLMQUIST, L. E., MATTERN, F., SCHIELE, B., ALAHUHTA, P., BEIGL, M., AND GELLERSEN, H.-W. Smart-its friends: A technique for users to easily establish connections between smart artefacts. In *Ubicomp 2001: Ubiquitous Computing* (2001), Springer, pp. 116–122.

[21] HUANG, K., DO, E.-L., AND STARNER, T. PianoTouch: A wearable haptic piano instruction system for passive learning of piano skills. In *Wearable Computers, 2008. ISWC 2008. 12th IEEE International Symposium on* (2008), IEEE, pp. 41–44.

[22] HWANG, I., CHO, J., AND OH, S. Privacy-aware communication for smartphones using vibration. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on* (2012), IEEE, pp. 447–452.

[23] JAMIESON, K., AND BALAKRISHNAN, H. PPR: Partial packet recovery for wireless networks. *ACM SIGCOMM Computer Communication Review 37*, 4 (2007), 409–420.

[24] KAAJAKARI, V., ET AL. Practical MEMS: Design of microsystems, accelerometers, gyroscopes, RF MEMS, optical MEMS, and microfluidic systems. *Las Vegas, NV: Small Gear Publishing* (2009).

[25] KRAWCZYK, M., AND GERKMANN, T. STFT phase reconstruction in voiced speech for an improved single-channel speech enhancement. *Audio, Speech, and Language Processing, IEEE/ACM Transactions on 22*, 12 (2014), 1931–1940.

[26] LESTER, J., HANNAFORD, B., AND BORRIELLO, G. Are You with Me?–Using Accelerometers to Determine If Two Devices Are Carried by the Same Person. In *Pervasive computing*. Springer, 2004, pp. 33–50.

[27] MARQUARDT, P., VERMA, A., CARTER, H., AND TRAYNOR, P. (sp) iPhone: decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 551–562.

[28] MAYRHOFER, R., AND GELLERSEN, H. Shake well before use: Intuitive and secure pairing of mobile devices. *Mobile Computing, IEEE Transactions on 8*, 6 (2009), 792–806.

[29] MILUZZO, E., VARSHAVSKY, A., BALAKRISHNAN, S., AND CHOUDHURY, R. R. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM, pp. 323–336.

[30] MORRIS, D., TAN, H. Z., BARBAGLI, F., CHANG, T., AND SALISBURY, K. Haptic feedback enhances force skill learning. In *WHC* (2007), vol. 7, pp. 21–26.

[31] NANDAKUMAR, R., CHINTALAPUDI, K. K., PADMANABHAN, V., AND VENKATESAN, R. Dhwani: secure peer-to-peer acoustic NFC. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (2013), ACM, pp. 63–74.

[32] NIWA, M., YANAGIDA, Y., NOMA, H., HOSAKA, K., AND KUME, Y. Vibrotactile apparent movement by DC motors and voice-coil tactors. In *Proceedings of the 14th International Conference on Artificial Reality and Telexistence (ICAT)* (2004), pp. 126–131.

[33] ROBERGE, J. K. *Operational amplifiers: theory and practice*. John Wiley & Sons, 1975.

[34] ROY, N., GOWDA, M., AND CHOUDHURY, R. R. Ripple: Communicating through physical vibration. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), pp. 265–278.

[35] RUMNEY, M., ET AL. *LTE and the evolution to 4G wireless: Design and measurement challenges*. John Wiley & Sons, 2013.

[36] SAXENA, N., AND WATT, J. H. Authentication technologies for the blind or visually impaired. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)* (2009), vol. 9, p. 130.

[37] SMARAGDIS, P., RAJ, B., AND SHASHANKA, M. Missing data imputation for spectral audio signals. In *Machine Learning for Signal Processing, 2009. MLSP 2009. IEEE International Workshop on* (2009), IEEE, pp. 1–6.

[38] STUDER, A., PASSARO, T., AND BAUER, L. Don't bump, shake on it: The exploitation of a popular accelerometer-based smart phone exchange and its secure replacement. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACM, pp. 333–342.

[39] TANNER, P., LOEBACH, J., COOK, J., AND HALLEN, H. A pulsed jumping ring apparatus for demonstration of Lenzs law. *American Journal of Physics 69*, 8 (2001), 911–916.

[40] TSE, D., AND VISWANATH, P. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[41] WAITE, A. D., AND WAITE, A. *Sonar for practising engineers*, vol. 3. Wiley London, 2002.

PhyCloak: Obfuscating Sensing from Communication Signals

Yue Qiao, Ouyang Zhang, Wenjie Zhou, Kannan Srinivasan and Anish Arora
Department of Computer Science and Engineering
The Ohio State University
$\{$*qiaoyu, zhouwe, kannan, anish*$\}$*@cse.ohio-state.edu*
zhang.4746@buckeyemail.osu.edu

ABSTRACT

Recognition of human activities and gestures using pre-existing WiFi signals has been shown to be feasible in recent studies. Given the pervasiveness of WiFi signals, this emerging sort of sensing poses a serious privacy threat. This paper is the first to counter the threat of unwanted or even malicious communication based sensing: it proposes a blackbox sensor obfuscation technique PhyCloak which distorts only the physical information in the communication signal that leaks privacy. The data in the communication signal is preserved and, in fact, the throughput of the link is increased with careful design. Moreover, the design allows coupling of the PhyCloak module with legitimate sensors, so that their sensing is preserved, while that of illegitimate sensors is obfuscated. The effectiveness of the design is validated via a prototype implementation on an SDR platform.

1 Introduction

A new form of threat has emerged recently that leaks private information about the whereabouts and activities of physical targets merely by observing the ongoing wireless communications in the scene. Broadly speaking, as a wireless signal gets reflected off of people and other objects in the scene, information about them is leaked to eavesdroppers by computational analysis of the signal distortions. Increasingly, researchers have been demonstrating proofs of concept where not only people presence but also fine-grain information about their locations and even breathing, lip movement or keystrokes is leaked [18, 28, 30, 1, 24]—all from observing communication signals that are widely prevalent in our homes. While the upside is that legitimate users can detect these physical "signatures" simply using existing signals, a burglar can also detect that there are no people in a house, a passerby can decipher key presses without leaving a trace [8], and a neighbor can snoop on the activities in our homes [30]. There is little doubt that several of these privacy exploits will in due course be realized robustly and commoditized for broad use. And, given the pervasive nature of wireless communications, the privacy implications of such attacks will undoubtedly be of major social importance.

It is thus timely and important to develop suitable counter-measures for this type of privacy leakage. We take the first step at tackling this problem by proposing a solution to address a single-antenna eavesdropping sensor. At first glance, it might appear that an obvious way to prevent or deter the privacy leakage is to simply jam the signals [21, 11]. However, jamming is an overkill for this problem, as the protection we wish lies in physical and not in the logical (data) layer. Jamming distorts the information of both layers, therefore it hurts the channel capacity of the network. In contrast to jamming, our approach is to distort the physical information that is environmentally superimposed on the signal as opposed to the data itself. *To make clear the distinction between these two forms of signal distortion, we refer to the latter as signal obfuscation.*

To avoid any modification of existing receivers, we need to build an obfuscator (Ox) that works independently from a receiver (Rx) and can yet deter privacy leakage against a single-antenna eavesdropper. At the same time, Ox should not hurt the ongoing reception at the intended receiver. In addition, given the diversity of the design of RF based sensors and invisibility of eavesdroppers, it is not reasonable to assume Ox that uses a specific obfuscation approach against a specific Eve. Thus, our goal is to build a black-box solution which distorts only the privacy sensitive information while not affecting the logical information. We design Ox by answering the two questions below:

1. How to distort physical information regardless of the RF-sensing mechanism? To answer this question, let us first examine what kind of physical information is contained in RF signals. Assume the received signal at a reflector is $s(t)$, then the received signal $r(t)$

reflected by the reflector can be expressed as follow: $r(t) = a \times s(t) \times e^{j2\pi(f_c+\Delta f)(t+\Delta t)}$, where a is the amplitude gain, f_c is the carrier frequency, Δf is the Doppler shift caused by a reflector that moves at a constant speed relative to the receiver, and Δt is the delay due to transmission over the path. Here, we can see that the reflector modifies the reflected copies by controlling three orthogonal components: amplitude gain a, delay Δt and Doppler shift Δf. All the features exploited by single-antenna RF based sensors are created by these three degrees of freedom (DoFs). Hence, if an Ox distorts the three orthogonal bases respectively, any features that reveal physical information are distorted too.

2. *How to preserve logical information (data communication)*? As the previous observation suggests, Ox needs to change the 3 degrees of freedom (DoFs) of a signal in order to deter eavesdropping of physically sensed features. Note that in a wireless environment, signals traverse through many paths and experience Doppler shifts: These effects are similar to dynamic multipath reflections. Thus, Ox can be a relay node that introduces dynamically changing multipath components of the communication signal. In other words, Ox receives the incoming communication signal, manipulates the signals and forwards them back to the environment. To a legitimate receiver, this forwarded signal will simply look like a multipath component of the signal from the legitimate transmitter (Tx). Commercial off-the-shelf (COTS) Rx is capable of tolerating and even exploiting multipath reflections to decode data. Thus, a carefully designed Ox can distort sensing and still preserve communication.

Challenges: PhyCloak works as a full-duplex amplify-and-forward (A&F) relay at logic layer, and an Ox at physical layer by distorting the 3 DoFs. While the solution may appear at first blush to be a simple instance of full-duplex A&F forwarder [6, 3], there are key challenges that arise from this design that need to be resolved.

1. *Online self-channel estimation with an ongoing external transmission*: Online self-channel estimation is needed for an Ox as it works in an environment where the channel is varying as a result of target movement, gestures and activities. When we combine the Ox module with a legitimate sensor the self-channel variation becomes more significant due to the moving object close to the sensor. Therefore an Ox has to transmit training symbols to acquire channel estimation every channel coherence interval (∼100ms). But a complication arises that the training needs to co-exist with ongoing data transmission. A straightforward way to overcome this problem is to adopt medium access control (MAC), however, that would introduce contention and hurt throughput of legitimate data transmission given the frequent self-channel updates.

2. *Effectiveness of obfuscating physical information*: No work has been done in validating a full-duplex A&F forwarder's capability of controlling physical information contained in the forwarded copy. In addition, the effectiveness of superposing an Ox's distorted signal and a target's reflected signal in obfuscating an eavesdropping sensor has yet to be shown.

Contributions: We propose PhyCloak to protect privacy information from unwanted or even malicious sensing with no modification to existing wireless infrastructures. In this work, we make the following contributions:

1. To our knowledge, we are the first to address the potential threats due to the recent development of communication-based sensing.

2. We propose PhyCloak, the first full-duplex forwarder-based solution that hides physical information superimposed by the channel via adding interference in a 3-dimensional orthogonal basis so that illegitimate sensing is disabled and meanwhile data transmission is not affected (and even improved). We go further and add the capability to spoof human gestures to further confuse illegitimate sensors.

3. We propose an alternative online self-channel estimation scheme that is contention-free and operates in the presence of an ongoing transmission. By doing so we also allow for legitimate sensing by integrating the sensor with our obfuscator.

4. We build a prototype PhyCloak on PXIe-1082, an SDR platform. Experimental results (Section 5.3) on a state-of-the-art sensor show that PhyCloak successfully obfuscates illegitimate sensing, enables legitimate sensing and improves overall throughput of data transmission. Gesture spoofing to the same type of sensor is also proved to be feasible.

2 Related Work

RF sensing from communications has been of great interest in the last few years, as it allows data signals to be exploited to infer remarkable details about the physical world. Although the primary purpose of the communication signals is to carry logical information, concepts of radar analysis [14, 5, 23, 16, 15, 25, 27, 22, 19, 26, 10, 13, 17] are adapted to extract these details. There are however several challenges in the adaptation since communication signal is defined particularly for carrying data. For example, radar systems control their resolution by specially encoding their transmitting signals, say in the form of Frequency-Modulated Carrier Waves (FMCW) for spectrum sweeping, but when sensing from RF communication a similar sort of transmitter cooperation typically cannot be leveraged. As another example,

Existing Work	Feature Basis	Device	Sensing Task
WiSEE: Pu et al. [24]	Doppler Shift	USRP-N210	Gesture recognition
Wi-Vi: Adib and Katabi [1]	Phase	USRP-N210	Gesture based communication,tracking
E-eyes: Wang et al. [30]	RSSI, CSI	COTS 802.11n devices	Activity classification
Gonzalez-Ruiz et al. [12]	RSSI	IEEE 802.11g wireless card	Obstacle mapping
Wang et al. [29]	Phase, CSI	COTS 802.11ac devices	Activity classification
WiKey: Ali et al. [2]	CSI	COTS 802.11n devices	Key recognition
RSA: Zhu et al. [32]	RSS	HXI Gigalink 6451 60GHz radios	Object imaging

Table 1: Summary of recent SISO sensing systems

sophisticated radar signal processing techniques, say creating a synthetic aperture using a *large* number of antennas, cannot be implemented directly in communication systems due to resource limitations.

Many techniques have been developed and demonstrated to address the above mentioned challenges for diverse sensing tasks including motion tracking [1], activity/gesture recognition [24, 30, 29], and obstacle/object mapping/imaging [32, 12], and even minor motions like keystrokes recognition [8, 2] and lip reading [28]. One idea is to use one antenna to emulate an antenna array in the presence of human movement. By tracking the angle of the reflected signal from the target (human) [1], the system is able to track the motion of the target as a form of inverse synthetic aperture radar (ISAR). Ubicarse [18] exploits the idea of circular synthetic aperture radar (SAR), in which the system rotates a single antenna so as to emulate a circular antenna array. As SAR does not require the target to be in motion, unlike the case of ISAR, Ubicarse proposes a method of using a handheld device to create circular antenna array to perform localization. To overcome any imprecision in the circle created by the rotation, it refines the formulation of SAR by using the relative trajectory between two receive antennas. Some other techniques characterize signatures corresponding to the channel variation caused by human activities. E-eyes [30] shows that temporal RSS and CSI features, which are available in COTS devices, can be used in activity classification, albeit this requires relatively heavy training. WiSee [24] proposes a method to extract Doppler shifts from OFDM symbols by applying a large FFT over repeated symbols, and gesture recognition is then shown to be possible from the extracted Doppler shifts. Another interesting technique used by communication based sensors maps obstacles/objects [12, 20]. The Tx-Rx pairs detect the presence of obstacles via wireless measurements and thereby co-operatively draw the indoor obstacle map.

As our protection system is single-input-single-output (SISO), we focus on breaking any SISO illegitimate sensing system in this work. Although SISO sensing systems use diverse techniques exemplified in Table 1, they all leverage a subset of the 3 DoFs discussed in Section 1. Since PhyCloak provides a generic tool to obfuscate in all these three dimensions, it can protect against any SISO sensor.

In contrast, for a multi-antenna sensing system, there is an additional DoF—the relative placement of antennas—that yields other types of information like angle of arrival (AoA) and time difference of arrival (TDoA). Nevertheless, by rotating PhyCloak's transmit antenna or extending our framework to a multi-antenna protection system, we would have the freedom to also obfuscate the fourth dimension provided by a multi-antenna sensing system.

3 Overview

3.1 Threat Model

Assume there is an adversary who is interested in inferring physical information from a SISO wireless communication channel. The adversary may be active or passive, i.e., it can transmit itself or just exploit ongoing wireless transmissions. In both cases, we assume that the adversary uses a single-antenna receiver to sniff the wireless transmission. In general, the design and implementation of adversarial sensing is unknown to the protection system designer.

Note that some types of sensing require a training phase to tune recognition patterns with respect to the environment of interest. To protect against stronger adversaries, we assume that the adversary is well trained for the environment at hand. The details of this training, whether it occurs concurrently with the training of a legitimate sensor or is based on some historical knowledge, are outside the scope of our interest here.

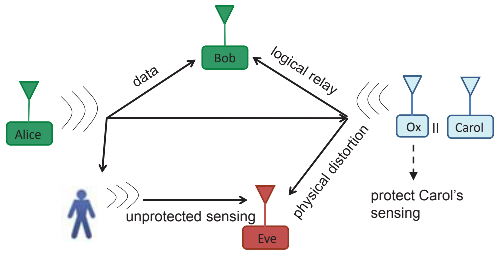


Figure 1: 4 single-input-single-output (SISO) nodes exist in the system: Alice, Bob, Carol and Eve: Alice and Bob perform data transmission and reception; Eve performs illegitimate sensing by exploiting Alice's transmission; Carol also performs sensing, but her obfuscator module forwards the received signal in a way that distorts physical information but preserves logical information

3.2 System and Goals

Our protection system comprises 4 SISO nodes as shown in Figure 1: Alice (data transmitter), Bob (data receiver), Carol (legitimate sensor) and Eve (illegitimate sensor). Both Alice and Bob can be controlled by Eve, thus Carol does not assume that Alice and Bob are honest.

Goals: 3 tasks co-exist in the network: data transmission between Alice and Bob, illegitimate sensing at Eve and legitimate sensing at Carol. By adding Ox to Carol with no cooperation from any of the other nodes, the protection system must satisfy the following three goals:

1. Obfuscate Eve's sensing.
2. Preserve Carol's sensing.
3. Not degrade the throughput of the link between Alice and Bob, nor introduce extra computation at Alice and Bob; i.e., Alice's and Bob's behaviors stay unaltered when Ox operates.

3.3 Three Degrees of Freedom

Usually a forwarder relays the signal directly, but in the context of an Ox a forwarder can do far more. In fact, a forwarder can be viewed as a special type of reflector; in theory, whatever change a natural reflector can induce on a signal, a forwarder can induce likewise. We begin by examining how a reflector changes the signal.

Letting the received signal at a reflector be $s(t)$, the received signal $r(t)$ that it reflects can be expressed as

$$r(t) = a \times s(t) \times e^{j2\pi(f_c+\Delta f)(t+\Delta t)} \tag{1}$$

where a is the amplitude gain due to reflection and propagation, f_c is the carrier frequency, Δf is the Doppler shift caused by a reflector that moves at a constant speed relative to the receiver, and Δt is the delay due to propagation over the path. We see that a reflector modifies signals by changing three components: a, Δf and Δt. Namely reflectors enjoy three DoFs when modifying signals.

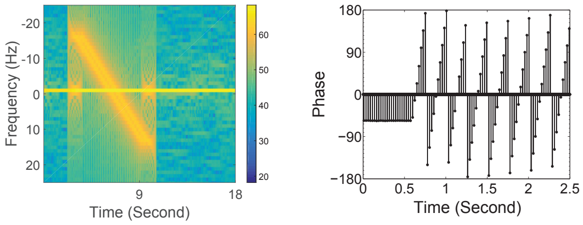
We examine what kind of signal processing is needed at the Ox to effect similar changes in the signal being forwarded. Rewrite Equation 1 into the following form:

$$r(t) = a \times s(t) \times e^{j2\pi\Delta f t} \times e^{j2\pi(f_c+\Delta f)\Delta t} \times e^{j2\pi f_c t} \tag{2}$$

Amplitude gain a: It is clear that if a forwarder receives $s(t)$ from the source, then by amplifying the samples with different levels, a can be easily changed.

Doppler shift Δf: To emulate a Doppler shift of Δf, a forwarder can rotate the nth received sample by $2\pi n\Delta f\overline{\Delta t}$, where $\overline{\Delta t}$ = sampling interval.

Delay Δt: A delay of Δt can be introduced by simply delaying the to-be-forwarded signals in either the digital domain or the analog domain at the forwarder. A problem with delaying signals in the digital domain is that digital delays are discrete and do not match the speed of human movement. For example, if an ADC works with a sampling rate 100MHz, then the minimum delay that can be introduced in digital domain is 10ns, which corresponds to a distance of 3m. Controlling analog delay while feasible, however requires effort in modifying existing SDR platforms. Our solution then is to rotate the to-be-forwarded samples by a fixed phase $2\pi(f_c+\Delta f)\Delta t$ in the digital domain, which matches the expected delay of Δt. In our NI PXIe platform, this calculation can be made in two clock cycles ($\frac{1}{\text{ADC sampling rate}}$).



(a) By multiplying the nth to-be-forwarded sample with $2\pi n\Delta f\overline{\Delta t}$, and changing Δf from 20Hz to -20Hz, the Doppler shift profile at the receiver is as expected

(b) By rotating the to-be-forwarded signals with a certain phase which changes by 36° every 30ms at the forwarder, the phase of the signal changes $\sim 36°$ every 30ms

Figure 2: Expected Doppler shift and phases are generated at a forwarder

Figure 2(a) depicts the Doppler shift profile of the received signals that are sent by a forwarder who keeps changing the to-be-forwarded samples' Doppler shift from 20Hz to -20Hz according to the above algorithm. Similarly, from Figure 2(b) we can see that by multiplying the to-be-forwarded samples with a phase φ which increases 0.2π every 30ms at the forwarder, the phase of the received samples changes by $\sim 0.2\pi$ every 30ms. These results show that a forwarder can predictably control Doppler shift and phase.

4 Design

Figure 3 shows a simplified block diagram of our system PhyCloak. The physical distortion is introduced after self-interference cancellation and then the distorted signal is then forwarded to the transmit antenna.

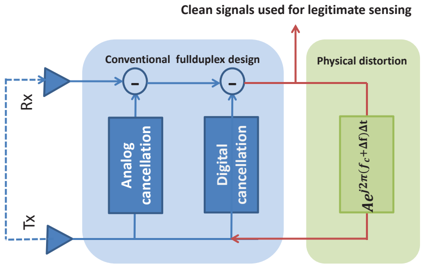

Clean signals used for legitimate sensing
Conventional fullduplex design
Physical distortion
Rx
Tx
Analog cancellation
Digital cancellation
$Ae^{j2\pi(f_c+\Delta f)\Delta t}$

Figure 3: High-level block diagram of PhyCloak

4.1 Online Maintenance of Self-Channel Estimates

As mentioned earlier, PhyCloak is a full-duplex system that needs to cancel self-interference to operate. However, human movement close to the full-duplex radio changes the self channel and affects cancellation. Figure 4 illustrates this phenomenon as it depicts the power of the residual noise after cancellation over time when a human target walks around the fulld-uplex radio. The full-duplex radio re-estimates the channel every 1s. We see that if we set the residual threshold to -95dBm, which is 5 dB above the maximum digital cancellation capability (noise = -100dBm), the channel estimation works fine only for a short duration (∼100ms) after each channel estimation update. This observation implies that frequent self-channel re-tuning (∼100ms) is required.

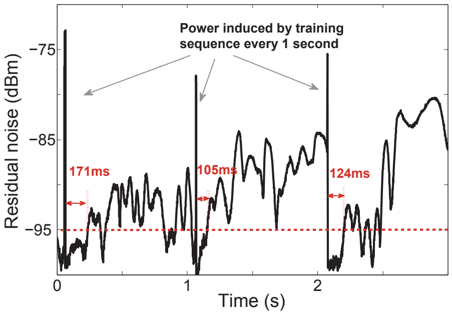


Figure 4: With human movement going on, the self-interference cancellatoin works fine only for a short duration (∼100ms)

A complication, however, arises when an update is attempted during an ongoing external transmission: the external transmission may distort self-channel estimation while the transmission that helps with self-channel estimation may interfere with external data reception. There are two straightforward solutions to this problem: 1) using MAC; 2) exploiting the silent period defined by wireless protocols, like short inter-frame space (SIFS) in WiFi. The former hurts the throughput of data transmission and moreover interrupted external transmission degrades coupling legitimate sensors with the Ox. And in addition, both of the solutions require a big effort to design careful adaptation to various wireless communication protocols.

We therefore propose a self-channel estimation algorithm for PhyCloak that addresses this complication. It uses two main elements: 1) oversampling and differential to get rid of any ongoing external transmission, and 2) a special training sequence that yields minimum interference to external transmissions.

4.1.1 Self-channel estimation with and without external interference

Before we describe our self-channel estimation algorithm, let us first see the impact of training with and without external interference. Assume $A = \{a_{-m}, a_{-m+1}, \ldots, a_m\}$ is the transmitted training sequence, $B = \{b_0, b_1, \ldots, b_m\}$ is the received sample sequence, and $H = \{h_0, h_1, \ldots, h_m\}$ is the channel coefficient vector in time domain with $m+1$ taps. Therefore, we have

$$\left\{\begin{array}{c} b_0 \\ b_1 \\ \cdots \\ b_m \end{array}\right\} = \left\{\begin{array}{ccc} a_0 & \cdots & a_{-m} \\ a_1 & \cdots & a_{-m+1} \\ \cdots & \cdots & \cdots \\ a_m & \cdots & a_0 \end{array}\right\} \times \left\{\begin{array}{c} h_0 \\ h_1 \\ \cdots \\ h_m \end{array}\right\} \qquad (3)$$

In the presence of external transmission, B becomes:

$$\left\{\begin{array}{c} b_0 \\ b_1 \\ \cdots \\ b_m \end{array}\right\} = \left\{\begin{array}{ccc} a_0 & \cdots & a_{-m} \\ a_1 & \cdots & a_{-m+1} \\ \cdots & \cdots & \cdots \\ a_m & \cdots & a_0 \end{array}\right\} \times \left\{\begin{array}{c} h_0 \\ h_1 \\ \cdots \\ h_m \end{array}\right\} +$$

$$\left\{\begin{array}{ccc} s_0 & \cdots & s_{-m} \\ s_1 & \cdots & s_{-m+1} \\ \cdots & \cdots & \cdots \\ s_m & \cdots & s_0 \end{array}\right\} \times \left\{\begin{array}{c} h'_0 \\ h'_1 \\ \cdots \\ h'_m \end{array}\right\} \qquad (4)$$

where $S = \{s_{-m}, s_{-m+1}, \ldots, s_i, \ldots, s_m\}$ is the external transmitted sample sequence, and $H' = \{h'_0, h'_1, \ldots, h'_m\}$ is the channel coefficient vector which corresponds to the channel between the transmit antenna of the external device and the receive antenna of the Ox.

4.1.2 Oversampling and differential to get rid of external interference

To overcome the external interference in Equation 4, which is unknown to PhyCloak, we exploit oversampling. Say PhyCloak samples at a rate $2m$ times higher than the sampling rate of the external transmitter, it follows that approximately $s_{-m} = \ldots = s_m$. So

$$\left\{\begin{array}{ccc} s_0 & \cdots & s_{-m} \\ s_1 & \cdots & s_{-m+1} \\ \cdots & \cdots & \cdots \\ s_m & \cdots & s_0 \end{array}\right\} \times \left\{\begin{array}{c} h'_0 \\ h'_1 \\ \cdots \\ h'_m \end{array}\right\} = \left\{\begin{array}{c} s_0 \times (h'_0 + \ldots + h'_m) \\ s_0 \times (h'_0 + \ldots + h'_m) \\ \cdots \\ s_0 \times (h'_0 + \ldots + h'_m) \end{array}\right\} \qquad (5)$$

Therefore, by differential we have

$$\left\{\begin{array}{c} b_1 - b_0 \\ b_2 - b_1 \\ \cdots \\ b_m - b_{m-1} \end{array}\right\} = \left\{\begin{array}{ccc} a_1 - a_0 & \cdots & a_{-m+1} - a_{-m} \\ a_2 - a_1 & \cdots & a_{-m+2} - a_{-m+1} \\ \cdots & \cdots & \cdots \\ a_m - a_{m-1} & \cdots & a_1 - a_0 \end{array}\right\} \times \left\{\begin{array}{c} h_0 \\ h_1 \\ \cdots \\ h_m \end{array}\right\} \qquad (6)$$

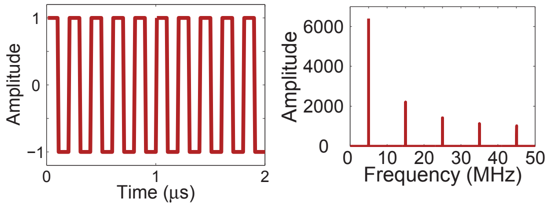
It may appear that we have already been able to get rid of external interference, however, $\mathbf{A}$ is an $m \times (m+1)$ matrix, so the rank of A is less than $m+1$. This means that we can get only a unique solution for at most m

of the $m+1$ unknowns contained in $\mathbf{H}$, where $\mathbf{H} = \{h_0, h_1, \ldots, h_m\}^T$ and

$$\mathbf{A} = \left\{ \begin{array}{ccc} a_1 - a_0 & \ldots & a_{-m+1} - a_{-m} \\ a_2 - a_1 & \ldots & a_{-m+2} - a_{-m+1} \\ \ldots & \ldots & \ldots \\ a_m - a_{m-1} & \ldots & a_1 - a_0 \end{array} \right\} \quad (7)$$

4.1.3 A special training sequence

To ensure that Equation 6 has a unique solution for $\{h_0, h_1, \ldots, h_{m-1}\}^T$, we leverage a special training sequence, namely a square wave, which is shown in Figure 5(a). As shown in Figure 5(b), the fundamental frequency of the square wave is the square wave frequency, and its odd harmonics are decreasing in size. To be more specific, for a square wave over a period consisting of N samples with B MHz sample rate, the frequency components are at $1f$, $3f$,..., $(2i+1)f$, ... with decreasing amplitude, where $f = \frac{B}{N}$MHz.



(a) Training sequence in time domain

(b) Training sequence in frequency domain

Figure 5: Training sequence

The rationale for using this training sequence is twofold: First, the square wave has a unique solution to $\{h_0, h_1, \ldots,\ h_{m-1}\}^T$ as long as $a_{-m} = a_{-m+1} = \ldots = a_0 = a_1 + c = \ldots = a_m + c$, where c is a non-zero constant. And second, the spikes it produces in the frequency domain are sparse. For example, with $B = 100$MHz and $N = 16$, the space between neighboring spikes is 12.5MHz. Such sparse spikes are tolerable in wireless systems. For example, in a 20MHz WiFi band using OFDM, as claimed by Flashback [9], existing WiFi systems have a relatively large SNR margin. And because the interference of any such spike is constrained to at most one subcarrier, the loss of a few bits does not significantly affect decoding, as successful packet transmissions always respect SNR margins.

4.1.4 The training procedure

Training is performed as follows: PhyCloak samples at a rate n times higher than that of external transmission. A training sequence which is the concatenation of consecutive 1s and -1s is sent during training. The received samples corresponding to the transition points (1 to -1 or vice versa) are used to calculate the channel coefficients. More specifically, the received sample b_0 which corresponds to the point right before the transition occurs is equal to $h_0 + \cdots + h_m$, and the next received sample b_1 is equal to $-h_0 + \cdots + h_m$. Thus, we can compute $h_0 = (b_0 - b1)/2$. The rest of the channel coefficients are calculated in a similar way. One concern is whether the desired oversampling rate can be supported. Take 802.11g as an instance, which has the smallest bandwidth (20MHz) among WiFi standards. If training were to require a 20X oversampling rate, we would need a platform that supports 400MHz sampling rate, which is very expensive. We figure out that, however, a 4X oversampling rate is sufficient to eliminate the effect of an external transmission of 802.11g. The reason is that the delay spread of non-ultra-wideband transmission in an indoor setting does not expand more than 3 taps.

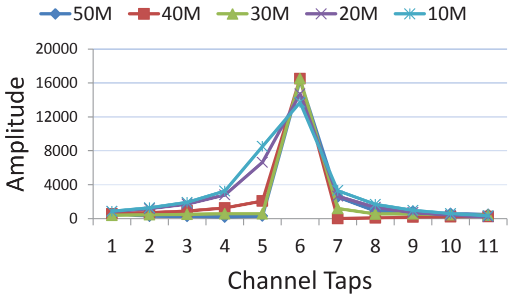
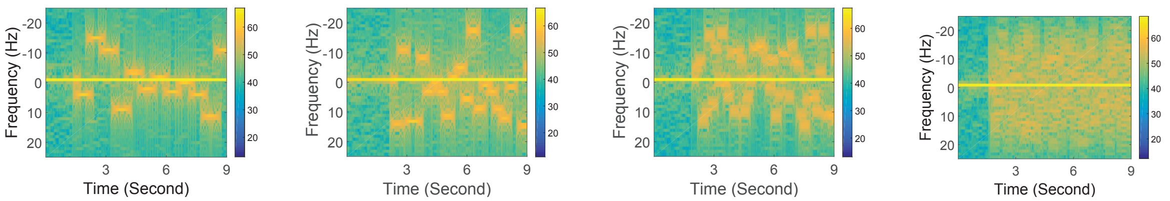


Figure 6: Channel coefficients measured at different sampling rates

To understand that, we need to know the fact that power delay profile is decided by two factors: multipath propagation and inter-symbol-interference (ISI). Let us study them one by one. First is the multipath propagation. For a 20MHz radio, one tap corresponds to $\frac{3\times10^8\text{m/s}}{20\text{MHz}} = 15$m. So the fourth tap corresponds to a 60-meter reflective path. The power conveyed by the 60-meter reflective path is significantly smaller than that conveyed by the short (∼10cm) line-of-sight path between the co-located transmitting and receiving antennas. Second, due to ISI each received sample is affected by not only the intended transmitted symbol, but also its two neighboring symbols. Therefore the delay spread expands across 3 taps. Figure 6 plots the channel estimation of the self channel under different sampling rates in the same environment. We see that in all cases, the main energy is always spread across 3 taps. So as long as we can accurately estimate the three dominant taps in non-ultra-wideband, we can achieve good cancellation performance. That implies we need the external interference to be stable during the reception of at least four consecutive samples at the transition point of the training sequence so as to get the three main taps by differential. Namely 4X oversampling is required.

Note that 4X oversampling does not guarantee the re-



(a) 1s (b) 0.5s (c) 0.125s (d) 0.05s

Figure 7: The granularity of the spectral decreases as the Doppler shifts change from 1s to 0.05s

ception of the desired 4 samples happen in the duration of one external interference sample. But we can leverage the interference reduction provided by averaging over multiple transition points, and partially accurate estimation of the channel taps, and still achieve good performance. Even lower oversampling rate (2X/3X) also performs well according to the experiment (see Section 5.2).

4.2 Obfuscation of Patterns in 3 DoFs

To motivate how we obfuscate patterns in the three DoFs, let us first examine the result of superposing a signal via one path with an obfuscated version via another path.

Assume we have two paths: one with $\{a_1, \Delta f_1, \Delta t_1\}$, and the other via the Ox with $\{a_2, \Delta f_2, \Delta t_2\}$. The superposition of the signals through these two paths is given by the following formula:

$$\begin{aligned}\hat{r}(t) =& a_1 \times s(t) \times e^{j2\pi(f_c+\Delta f_1)(t+\Delta t_1)} \\ &+ a_2 \times s(t) \times e^{j2\pi(f_c+\Delta f_2)(t+\Delta t_2)}\end{aligned} \tag{8}$$

Now, is superposing an obfuscated signal sufficient for hiding the original triplet $\{a_1, \Delta f_1, \Delta t_1\}$? The answer is partially yes: The amplitudes and delays are instantaneously covered in the superposed signal, but the respective Doppler shifts remain distinguishable after superposition. So, a and Δt can be hidden instantly by randomly changing amplitude and delay of the signal by the Ox.[1] To see why Doppler shifts are distinct even after superposition, consider the frequency response of the received signals:

$$\begin{aligned}R(f) =& \int \hat{r}(t) e^{-2\pi jft} dt \\ =& \int (a_1 \times s(t) \times e^{j2\pi(f_c+\Delta f_1)(t+\Delta t_1)}) e^{-j2\pi ft} dt \\ &+ \int (a_2 \times s(t) \times e^{j2\pi(f_c+\Delta f_2)(t+\Delta t_2)}) e^{-j2\pi ft} dt \\ =& a_1 e^{j2\pi(f_c+\Delta f_1)\Delta t_1} S(f - fc - \Delta f_1) \\ &+ a_2 e^{j2\pi(f_c+\Delta f_2)\Delta t_2} S(f - fc - \Delta f_2)\end{aligned} \tag{9}$$

where $S(f)$ is the frequency response of $s(t)$. In an OFDM system, we can see two frequency components that are shifted by Δf_1 and Δf_2 around the subcarrier f.

4.2.1 Doppler shift obfuscation

As amplitude and delay can be instantly changed by superposition with an obfuscated signal, patterns that rely only on amplitude and delay can be hidden by Ox, by randomly changing them on a per packet basis. At first glance, it may appear that this scheme cannot be made to work for patterns that rely on Doppler shift, but it turns out the scheme can be made to work for Doppler shift, assuming the moments of change are carefully chosen.

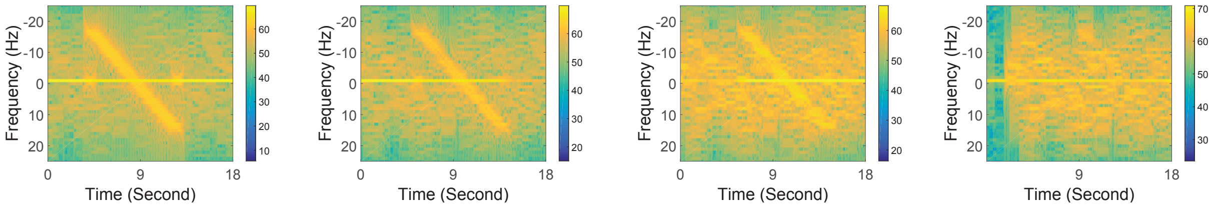
The rationale for choosing the moments of change is based on the fact that a t-second observation in the time domain leads to $1/t$ Hz granularity in the frequency domain. To choose the appropriate Δf at $1/t$ Hz granularity, there is an implicit requirement that the Δf needs to last for at least t seconds. Therefore, if the forwarder changes its Δf every t seconds while the other copy's Δf does not change, an observer would still only see $1/t$ Hz granularity. Since human movements typically result in -20Hz to 20Hz Doppler shifts in the 2.4GHz band, a Doppler shift of the forwarded copy that changes every 0.1s creates sufficient confusion at an observer. Figure 7 shows that when the Doppler shifts of the transmitted signals are varied from every 1s to every 0.05s, the spectral seen by an observer with 1s observation interval have progressively finer granularity, to the point where a time-frequency pattern gets hidden.

4.2.2 Effect of superposing with randomly changing obfuscated signals

The basic idea of PhyCloak then is to superpose signals from the target with naturally changing $\{a, \Delta f, \Delta\phi\}$ with the obfuscated signals with randomly changing $\{a, \Delta f, \Delta\phi\}$. More specifically, as analyzed above, PhyCloak changes the value of the triple every 0.1s. We illustrate the blackbox effect of obfuscation experimentally using two state-of-the-art sensors, WiSee [24] and Wi-Vi [1], which we implemented. WiSee performs gesture recognition by extracting Doppler shifts from OFDM symbols, whereas Wi-Vi uses ISAR to track the angle of human motion with respect to the receive antenna of the sensor.

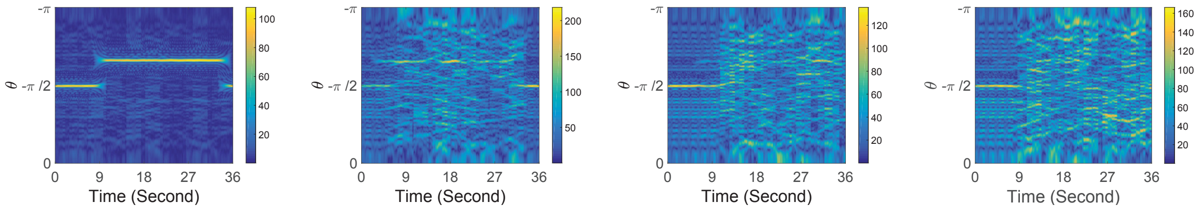
For the case of obfuscating Doppler shift patterns, Figure 8 shows the superposition of a signal with the syn-

[1] In theory for a high sampling rate receiver, delays might be separable in the brief prefix that arrives before the obsfuscated signal arrives, but how much information a sensor can accurately extract from the brief clean prefix is questionable.



(a) 0dB (b) -3dB (c) -6dB (d) -9dB

Figure 8: The pattern that a WiSee sensor sees in Figures 2(a) is hidden by an obfuscated signal where Doppler shift changes every 0.1 second



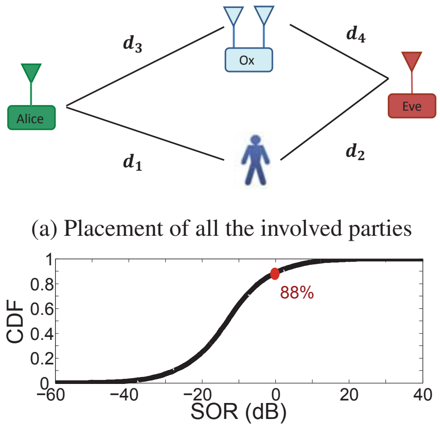
(a) Motion towards a Wi-Vi style sensor with constant angle (b) 0dB (c) -3dB (d) -6dB

Figure 9: The constant angle of human motion (starting from 9th second) that a Wi-Vi style sensor sees in (a) is hidden by an obfuscated signal where phase changes randomly every 0.1 second

thetically generated Doppler shift pattern described in Figure 2(a) and an obfuscated copy of the pattern where Doppler shift changes randomly every 0.1s. We see that pattern of Figure 2(a) is covered by the "noise map" created by the randomly changing copy. As the strength ratio of the former relative to the latter, which we define as signal to obfuscation ratio (SOR), decreases from 0dB to -9dB, the visibility of the artificial pattern decreases.

For the case of obfuscating phase-based patterns, we synthetically emulated a human moving towards the receive antenna of our Wi-Vi style sensor at a constant angle, as shown in Figure 9(a), and then superposed the signal with a randomly obfuscated copy where phase changes every 0.1s. Figure 9 shows that as SOR decreases from 0dB to -6dB, the pattern shown in Figure 9(a) becomes progressively invisible at the Wi-Vi style sensor.

It is worth noting that power passively reflected by human is much smaller compared to that actively forwarded by an Ox that has its own power supply. Therefore 0dB SOR can be readily achieved. To illustrate this point, we can build a simplified power model of our system. In our system Ox's goal is to minimize SOR at Eve with no knowledge of the locations of any of the other parties, so its best strategy is to work at the maximum transmission power. If we assume free-space attenuation, then SOR $\sim \frac{a}{A}(\frac{d_3 d_4}{d_1 d_2})^2$, where a and A are the reflection gains at target and Ox respectively, and d_1,d_2,d_3 and d_4 are the distances as shown in Figure 10(a). Figure 10(b) plots the simulation result of the CDF of SOR when we randomly place Alice, Eve, Ox and human target in a 10m×5m room, with reflection gains being set to -3dB and 10dB respectively. We see that in around 88% cases, SOR is smaller than 0dB.



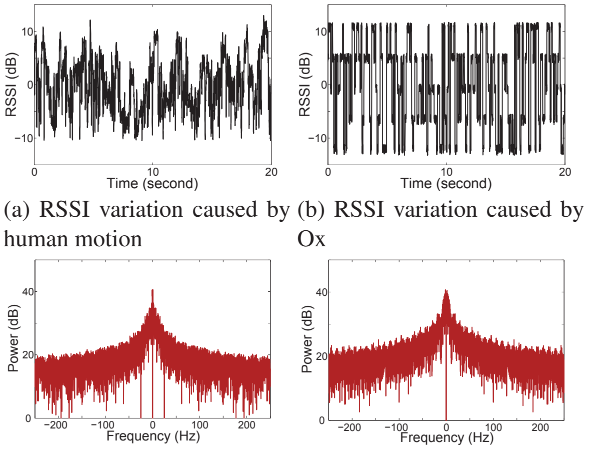
(a) Placement of all the involved parties

(b) SOR distribution

Figure 10: A simplified power model

4.2.3 Security analysis

We believe that our system is robust against a single antenna eavesdropper given certain SOR because of the fact: little information can be extracted from two random signals which occupy similar bands as long as the power of the undesired signal is higher than that of the desired one. In our case, the desired signal is the natural channel variation induced by target, while the undesired one is the artificial channel variation induced by PhyCloak. It is worth noting that as human motion is slow, natural channel variation has a small bandwidth, which is comparable to that of the artificial channel variation that changes ev-



(a) RSSI variation caused by human motion

(b) RSSI variation caused by Ox

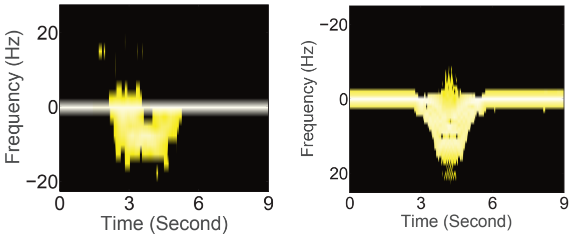
(c) Power spectrum of the RSSI trace in (a)

(d) Power spectrum of the RSSI trace in (b)

Figure 11: The signal (real channel state information) and the noise (artificial channel state information) have similar bandwidths

ery 0.1s.

To illustrate the above point, we compare the RSSI variations induced by human and PhyCloak. Figure 11(a) and 11(b) plot the RSSI changes caused by human movement and PhyCloak respectively, and Figure 11(c) and 11(d) plot the corresponding power spectrums. From the figure, we see that the occupied bandwidths of the two channel state traces are similar.



(a) Doppler shifts created by a human gesture (pull)

(b) Doppler shifts emulated by an Ox

Figure 12: Spoofing

4.3 Spoofing

According to the above discussion, our design succeeds in obfuscating any RF-based single-antenna sensors by creating false negative results. But an Ox can achieve more than that: it can create false positives also by spoofing changes in the 3 DoFs that are similar to the changes created by a target. By splitting the to-be-forwarded samples into multiple streams, applying different instantiations of the triple $\{a, \Delta f, \Delta t\}$ to them, and forwarding the combination of the processed streams as one stream, an Ox can emulate multiple reflectors corresponding to different parts of the target (say a human body). But unlike the case of false negatives, the effectiveness of creating false positives at a sensor grows as the Ox knows more about the features and algorithms used by the sensor. For example, if an Ox knows a sensor uses the WiSee algorithm [24], it can create a Doppler shift profile accordingly without making an effort to model accurate human movement. Figure 12 depicts the extracted Doppler profile of a human gesture (pull) and that spoofed by an Ox. WiSee segments a Doppler profile into positive and negative parts according to its power distribution and encodes them into 1s and -1s respectively. Since both of the profiles contain positive Doppler shifts of negligible power, they will be encoded as -1s and mapped to the same target by a WiSee sensor.

4.4 PhyCloak

By obfuscating using random physical distortion, an Ox is able to confuse Eve, and by online maintenance of self-channel estimates, Ox is able to output interference-free signals to Carol for legitimate sensing. However, one critical requirement is still not met: preserving the communication throughput in the presence of Ox.

Although PhyCloak works as a relay at logical layer which can potentially improve the throughput [31], it is not clear that obfuscation would not hurt the decoding process. We find that, however, as long as the change of the triplet $\{a, \Delta f, \Delta\phi\}$ does not happen in the middle of packet transmission, obfuscation is safe with respect to data communication. The reason for this is that from the perspective of a data receiver, the Ox effectively just adds variability to the channel. Since data receivers usually perform channel estimation at the beginning of the received packet, as long as the channel is stable during the reception of the packet, decoding can be successful. We, therefore, refine the design of PhyCloak as follows: PhyCloak switches between two transmitting modes: training and forwarding. In the training phase, the PhyCloak sends the above mentioned training sequence and computes its self-channel estimate according to Section 4.1.4; in the forwarding phase, PhyCloak then performs self-interference cancellation, applies the physical distortion $\{a, \Delta f, \Delta\phi\}$ to the interference-free signal and forwards the distorted signal via the transmit antenna. The PhyCloak randomly chooses an instance of $\{a, \Delta f, \Delta\phi\}$ in the predefined pool and updates the current value when the channel is free and the last update happened more than 0.1s ago. In this way, PhyCloak avoids interfering with the transmission. And in theory, there is still a chance that due to the delay caused by free-channel detection, PhyCloak changes the channel after several samples of a packet has been transmitted, but that chance is quite low. Even if it happens, because PhyCloak only affects a few samples at the beginning, the packet might still be decodable.

5 Validation

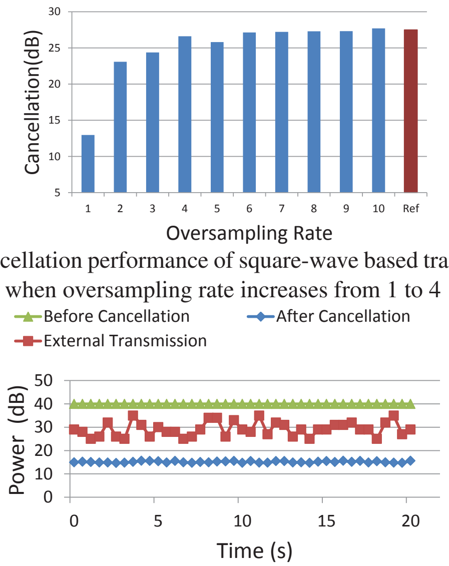
We now describe a prototype of PhyCloak that we have built, and our experiments to validate its performance.

5.1 Experimental Setup

Our prototype is based on PXIe 1082 SDR platform. We built the transmitter, receiver, eavesdropping sensor and legitimate sensor on the same platform, which all follow the 802.11g standard, i.e., working at 2.4GHz with a 20MHz band. PhyCloak works at the same center frequency but with a 50MHz sampling rate, about 3 times the rate of an external data transmission, which gives it a reasonable margin to perform self-channel estimation with an ongoing external transmission (see Section 5.2).

PhyCloak contains two RF chains, one for transmitting and one for receiving. Each of the RF chains contains an NI-5791 (FlexRIO RF transceiver equipped with one antenna) for transmitting or receiving and an NI PXIe-7965R (a Xilinx Virtex-5 FPGA) for digital processing. Analog cancellation is implemented according to our earlier design [7, 4]. The self-channel estimation, digital cancellation and physical layer distortion are implemented on the FPGA. The distortion processing introduces a latency of about 100ns. Our experiments were conducted in a 5m×7m lab.

5.2 Self-Interference Cancellation



(a) Cancellation performance of square-wave based training increases when oversampling rate increases from 1 to 4

(b) Insensitivity of square-wave based training to external transmission power variation, which is necessary for preserving legitimate amplitude based sensing

Figure 13: Self-interference cancellation performance

We begin with the performance of the digital cancellation of our self-channel estimation algorithm. As discussed in Section 4.1.4, Ox tolerates external interference during self-channel estimation using oversampling. So, we first examine the oversampling rate needed to achieve reasonably accurate self-channel estimates in the presence of external transmission. We let a full-duplex transceiver operate at 50MHz with a 10-tap filter for self-interference (digital) cancellation. Self-channel estimation is obtained by averaging over 128 training rounds, which altogether takes about 20μs.

Figure 13(a) plots the self-interference cancellation performance of our square-wave based training. In the figure, as we fixed the sampling rate of the full-duplex radio (50MHz), different oversampling rates correspond to different external transmission rates with the received power of the external transmissions being the same as that of self-interference signal at Ox's receive antenna[2]. 1X oversampling rate corresponds to the case when the training and data communication use the same sampling rate, in which case square-wave based training and traditional pilot based training would achieve similar performance. We see that the performance of self-interference cancellation of square-wave based training gets better as the oversampling rate increases from 1 to 4, but it stops increasing after 4, and achieves similar performance as that in the case when there is no external transmission going on (indicated by the red bar). It shows that Ox can reliably estimate and cancel self-interference even in the presence of strong external transmission when the oversampling parameter is 4X as supported by our observation in Section 4.1. In addition, 2X and 3X oversampling rates also produce high cancellation as they benefit from two factors: 1) accurate estimation of part of the channel taps, and 2) averaging over multiple transition points. **Takeaway:** *Our oversampling technique makes self-interference cancellation reliable at modest oversampling rates even in the presence of strong ongoing external transmission.*

The analysis above considers external interference sent at a fixed power. To enable legitimate sensing, self-interference cancellation performance needs to be stable even when the received power from external transmission is varying. For example, an unstable self-interference canceler can render an amplitude-based sensor useless since the (varying) residual self-interference will affect the received signal amplitude. Figure 13(b) plots the full-duplex radio's cancellation performance with 3X oversampling rate over time during which the received power from the external transmitter fluctuates. We see that the self-interference cancellation performance of square-wave based training is insensitive to the variation of external interference. **Takeaway:** *Our oversampling technique results in a stable cancellation performance at*

[2] Note that this is a very strong external interference and we choose this setting to show oversampling strategy's performance even under strong external interference.

modest oversampling rates even when the received signal from external transmitter is varying.

5.3 Obfuscation Performance

5.3.1 Obfuscation vs. SOR in 3 DoFs

We first measure the different levels of obfuscation created by PhyCloak by comparing the correlation of the amplitude, phase and Doppler shift with and without the presence of PhyCloak. The transmitter is programmed to send continuous OFDM symbols with QPSK modulation with varying amplitude and phase. An artificial Doppler shift of 10Hz is also added at the transmitter. PhyCloak performs obfuscation by randomly changing the amplitude, phase and Doppler shifts every 0.1s.

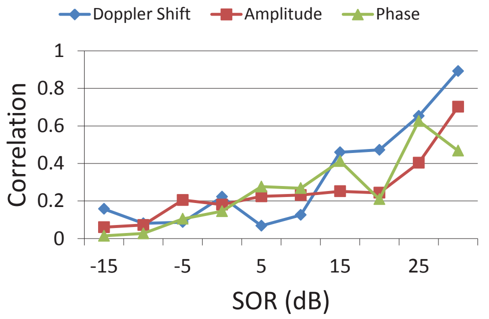


Figure 14: Obfuscation level of each of the three features decreases as SOR (original signal over obfuscation signal) increases

Figure 14 depicts the correlation between the pairs of amplitude, phase, and Doppler shifts at different SORs: Again, SOR is the signal strength ratio of original signal over obfuscation signal (see Section 4.2.2). We see that as SOR increases, the correlation of each pair of the three features increases, i.e., the obfuscation degree decreases. Amplitude sequence pair and phase sequence pair see lower correlation than Doppler shift pair when SOR is high. This is because amplitude and phase are instantaneous quantities, while Doppler is a statistical quantity that is derived from multiple instantaneous samples. But, even for Doppler shift, a 10dB SOR is low enough to hide the patterns contained in signals reflected by targets. It's worth noting that in practice, as PhyCloak is independently powered while the target only passively reflects signals, the desired SOR to successfully obfuscate is readily achieved. **Takeaway:** *PhyCloak effectively obfuscates sensing even at a relatively high SOR.*

As different sensors differ in their robustness to noise, PhyCloak's effectiveness is sensor dependent. While we are unaware of any research on the robustness of the communication-based sensors, we may infer from Figures 8, 9 and 14 that less obfuscation power is needed to confuse a phase or amplitude based sensor as compared to a Doppler shift based sensor. Therefore, we choose to validate the PhyCloak's capability of confusing illegitimate sensing and preserving legitimate sensing in the context of WiSee, which is the state-of-the-art Doppler shift based sensor.

5.3.2 Degradation of illegitimate sensing

We built a Doppler-based sensor in our platform per the method proposed by WiSee [24]. The method consists of two parts: 1) extraction of Doppler shifts from repeated OFDM symbols by applying a large size FFT; and 2) using sequence matching to classify gestures. We note since we could not get to the original WiSee code and some of the details are missing, we implement WiSee with a few adaptations. For example, we randomly map the sequence to the predefined classes with uniform distribution in case the sequence does not match any of the predefined sequence. Our implementation shows a classification accuracy of 93% across 5 gestures in none-line-of-sight (NLoS) setting with the human target 5 feet away from the WiSee sensor, while WiSee reports 94% across 9 gestures. While there is this small discrepancy in replication, the core algorithm is the same and our main goal is to study obfuscation performance.

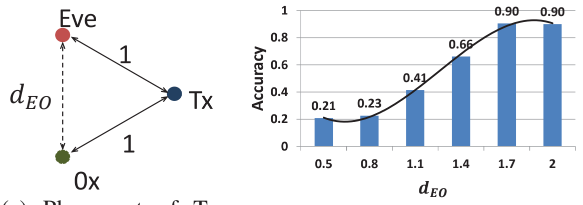
We examine the performance of an illegitimate WiSee sensor with obfuscation from a PhyCloak. We conduct two sets of experiments to validate PhyCloak's coverage range and its overall effectiveness under different channel conditions respectively.

Obfuscation coverage: First, we randomly choose 10 pairs of locations to place Tx and Eve, and then place Ox in locations such that the distance d_{TE} between Tx and Eve is equal to the distance d_{TO} between Tx and Ox as shown in Figure 15(a), but the distance d_{EO} between Eve and Ox varies from $0.5d_{TE}$ to $2d_{TE}$. The channels between any two of the three parties are line-of-sight (LoS).[3] A human target performs five gestures drag, push, pull, circle and dodge close to Eve. With no obfuscation, Eve's classification accuracy in this placement is about 90% across the five gestures.

For simplicity, we normalize d_{EO} by d_{TO} (d_{TE}), and plot the classification accuracy against the normalized d_{EO} in Figure 15(b). As we know, the received obfuscation power at Eve from Ox is a function of d_{TO} and d_{OE}, therefore as d_{EO} increases the power ratio of obfuscation over human reflection decreases. From the figure we see that classification accuracy of Eve increases as d_{EO} increases as expected. Note that since we have 5 classes,

[3] WiSEE sensors have a slightly worse performance in LoS ($\approx$ 90%) than NLoS ($\approx$ 93%) as strong direct power from the transmitter hides the information provided by target's reflection. For the next two experiments, we choose LoS instead of NLoS because it makes the placement easier to make sure d_{EO} is the only variable which would change the power ratio of the obfuscation and human reflection.

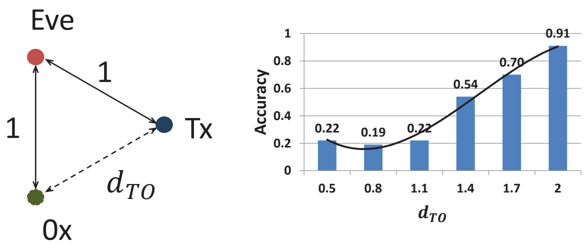
a classification accuracy of 0.2 means a random guess. PhyCloak can obfuscate Eve near perfectly when d_{EO} is smaller than 0.8, and it totally fails when it is larger than 1.7. **Takeaway:** *The closer Ox is to Eve, the better the achieved obfuscation.*



(a) Placement of Tx, Ox and Eve with all three channels LoS

(b) Classification accuracy of Eve in the presence of PhyCloak increases as d_{EO} increases

Figure 15: Eve's classification accuracy vs d_{EO}



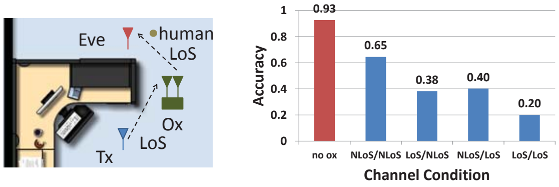
(a) Placement of Tx, Ox and Eve with all three channels LoS.

(b) Classification accuracy of Eve in the presence of PhyCloak increases as d_{TO} increases

Figure 16: Eve's classification accuracy vs d_{TO}

In the second experiment, we make $d_{TE} = d_{OE}$, and vary d_{TO} as shown in Figure 16(a). And again in Figure 16(b), we see that as d_{TO} increases, Eve's classification accuracy increases. **Takeaway:** *the closer Ox is to Tx, the better obfuscation is achieved.*

In other experiments we vary either the human-Eve or human-Ox distance while keeping the power received by Ox and human from Tx stay constant. As these distances respectively reduced, the effectiveness of the sensing and obfuscation respectively increased.



(a) Example of LoS/LoS placement in the lab

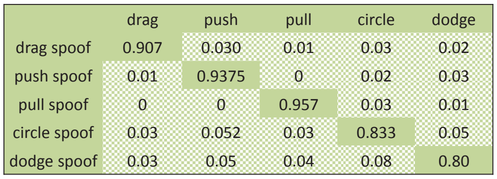
(b) Obfuscation degrades somewhat if Tx-Ox or Ox-Eve channels are NLoS

Figure 17: Eve's classification accuracy under different Tx-Ox and Ox-Eve channel conditions

Obfuscation effectiveness under different channel conditions: In addition to the coverage range in LoS setting, we also measure Eve's classification accuracy when channels between the transmitter and obfuscator and the channel between the obfuscator and Eve are under different LoS and NLoS combinations. Intuitively, when both channels are NLoS, Eve receives the least power forwarded by the obfuscator, and therefore, she achieves the best performance. We care about these channel conditions because in some scenarios the transmitter is under control of the adversary, and therefore the adversary may enjoy the freedom to create "good" channels to mitigate PhyCloak's obfuscation.

In the experiment, we make the channel between Tx and Eve NLoS, and the channel between Tx and the human and that between human and Eve LOS, so as to make sure Eve sees high classification accuracy when no obfuscation is going on. The channel between Tx and Ox and the channel between Ox and Eve have four possible channel condition combinations. A human target performs 500 times of the 5 predefined gestures near Eve in each of the four combinations. Figure 17(a) is an example of how we create a channel combination of Los/Los in the lab, where the first LoS refers to the channel condition of the channel between Tx and Ox, while the second refers to that of the channel between Ox and Eve. NLoS channels are created by placing obstacles in the direct propagation paths.

Figure 17(b) depicts Eve's classification accuracy without obfuscator and with obfuscator in four channel combinations. We can see that as expected, Eve sees the highest classification accuracy (65%) in NLoS/NLoS setting among the four channel conditions, but it is still smaller than the case when no obfuscation is happening (93%). Eve sees similar performance in Los/NLoS and NLoS/LoS scenarios as power forwarded by obfuscator in both the settings is similar. **Takeaway:** *although NLoS channel degrades the received power at Eve from Ox, the degradation is not dramatic since there is rich multipath propagation in indoor environment.*



	drag	push	pull	circle	dodge
drag spoof	0.907	0.030	0.01	0.03	0.02
push spoof	0.01	0.9375	0	0.02	0.03
pull spoof	0	0	0.957	0.03	0.01
circle spoof	0.03	0.052	0.03	0.833	0.05
dodge spoof	0.03	0.05	0.04	0.08	0.80

Figure 18: False positives with a spoofing Ox

5.3.3 Feasibility of spoofing

We built a spoofing obfuscator by reverse engineering the five predefined sequences corresponding to the five gesture types that our WiSee sensor recognizes. The basic difference between this spoofing obfuscator and PhyCloak is that the former changes Doppler shift according to the five well-defined gestures, while the latter changes

Doppler shift randomly. The result is shown in Figure 18. **Takeaway:** *the spoofing obfuscator fools a WiSee sensor with a high success rate, averaging 88.69% across the 5 gestures, in the absence of human gesturing*.

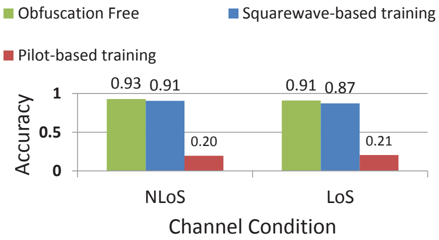


Figure 19: Square wave based training preserves legitimate sensing

5.3.4 Preservation of legitimate sensing

Next, we examine PhyCloak's capability of supporting coupled legitimate sensing. That is, we evaluate whether our self-channel estimation method produces consistent and sufficient self-interference cancellation in a changing environment to preserve legitimate sensing. Figure 19 depicts the legitimate sensor's classification accuracy for three different sensing modes: 1) obfuscation free sensing; 2) legitimate WiSee sensing coupled with a PhyCloak module that uses the proposed square-waved based self-channel estimation; 3) legitimate WiSee sensing coupled with a PhyCloak module that uses traditional pilot based self-channel estimation. We also vary the channel between Tx and the legitimate sensor by placing and removing obstacles. From the figure we see that the WiSee sensor equipped with PhyCloak module that uses square-wave based training achieves comparable performance as obfuscation-free sensing in both LoS and NLoS, while the WiSee sensor equipped with PhyCloak module that uses traditional training fails dramatically. This is because not enough self-interference cancellation is achieved in the presence of external transmissions using extant self-channel estimation techniques. **Takeaway:** *Square wave based training provides sufficient self-interference cancellation to preserve legitimate sensing with external transmission going on*.

5.4 Throughput Performance

As discussed in Section 4.4, PhyCloak would not hurt the average throughput by virtue of being a relay as long as it avoids parameter changes in the middle of packet transmissions. And, its online training would introduce some interference albeit of small measure. To validate that the net throughput benefit that a data receiver obtains from PhyCloak is not affected but even improved, we measured the throughput performance of a data link with and without PhyCloak in our testbed. We randomly picked 20 location triples to place a data transmitter, a data receiver, and PhyCloak. The data transmitter transmits back-to-back packets continuously, and we can thus see the throughput performance in the worst case where PhyCloak performs parameter updates in the middle of some packets. Figure 20 plots the CDF of the throughput with and without the PhyCloak. **Takeaway:** *The average throughput increases with the help of PhyCloak*.

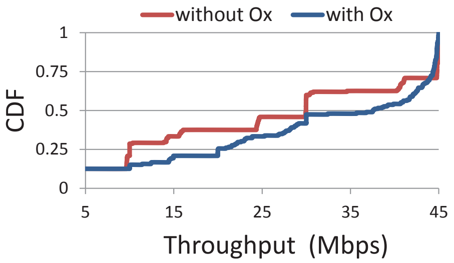


Figure 20: Throughput

6 Conclusion

We have shown that the threat created by recent developments in communication based sensing can be countered in a black-box fashion. PhyCloak obfuscates multi-dimensional physical signatures of human targets. We have empirically validated this for certain state-of-the-art sensors. We have also shown that when white box details of particular sensors can be obtained, PhyCloak can be refined to spoof those sensors. Notably, the methodology not only preserves but in fact improves the link throughput of the ongoing data transmissions, and supports co-existence of legitimate sensors while obfuscating illegitimate sensors.

Looking beyond the scope of the present work, we find that the methodology is readily generalized to protect against sensing of other types of physical targets and their properties, and allows for a network of PhyCloak devices to collaboratively cover a large region, the details of which are topics for future studies. In addition, when we extend our current single-antenna PhyCloak to a multiple-antenna system, how to fully exploit the space diversity provided by the multiple antennas is worth studying.

References

[1] Fadel Adib and Dina Katabi. *See through walls with WiFi!*, volume 43. ACM, 2013.

[2] Kamran Ali, Alex Xiao Liu, Wei Wang, and Muhammad Shahzad. Keystroke recognition using wifi signals. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 90–102. ACM, 2015.

[3] Dinesh Bharadia and Sachin Katti. Fastforward: fast and constructive full duplex relays. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 199–210. ACM, 2014.

[4] Dinesh Bharadia, Emily McMilin, and Sachin Katti. Full duplex radios. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 375–386. ACM, 2013.

[5] Igal Bilik and Joseph Tabrikian. Radar target classification using doppler signatures of human locomotion models. *IEEE Transactions on Aerospace and Electronic Systems*, 43(4):1510–1522, 2007.

[6] Bo Chen, Yue Qiao, Ouyang Zhang, and Kannan Srinivasan. Airexpress: Enabling seamless in-band wireless multi-hop transmission. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 566–577. ACM, 2015.

[7] Bo Chen, Vivek Yenamandra, and Kannan Srinivasan. Flexradio: Fully flexible radios and networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 205–218, 2015.

[8] Bo Chen, Vivek Yenamandra, and Kannan Srinivasan. Tracking keystrokes using WiFi. In *Proceedings of ACM MobiSys*, 2015.

[9] Asaf Cidon, Kanthi Nagaraj, Sachin Katti, and Pramod Viswanath. Flashback: decoupled lightweight wireless control. *ACM SIGCOMM Computer Communication Review*, 42(4):223–234, 2012.

[10] Theodoros Damoulas, Jin He, Rich Bernstein, Carla P Gomes, and Anish Arora. String kernels for complex time-series: Counting targets from sensed movement. In *2014 22nd International Conference on Pattern Recognition (ICPR)*, pages 4429–4434. IEEE, 2014.

[11] Shyamnath Gollakota, Haitham Hassanieh, Benjamin Ransford, Dina Katabi, and Kevin Fu. They can hear your heartbeats: non-invasive security for implantable medical devices. *ACM SIGCOMM Computer Communication Review*, 41(4):2–13, 2011.

[12] Alejandro Gonzalez-Ruiz, Alireza Ghaffarkhah, and Yasamin Mostofi. An integrated framework for obstacle mapping with see-through capabilities using laser and wireless channel measurements. *Sensors Journal, IEEE*, 14(1):25–38, 2014.

[13] Jin He and Anish Arora. A regression-based radar-mote system for people counting. In *International Conference on Pervasive Computing and Communications (PerCom), 2014 IEEE*, pages 95–102, March 2014. doi: 10.1109/PerCom.2014.6813949.

[14] Chih-Wei Huang and Kun-Chou Lee. Application of ica technique to pca based radar target recognition. *Progress In Electromagnetics Research*, 105:157–170, 2010.

[15] Youngwook Kim and Hao Ling. Human activity classification based on micro-doppler signatures using an artificial neural network. In *Antennas and Propagation Society International Symposium, 2008. AP-S 2008. IEEE*, pages 1–4. IEEE, 2008.

[16] Youngwook Kim and Hao Ling. Human activity classification based on micro-doppler signatures using a support vector machine. *IEEE Transactions on Geoscience and Remote Sensing*, 47(5):1328–1337, 2009.

[17] Vinit Kizhakkel, Rajiv Ramnath, and at el. Pulsed doppler radar target recognition based on micro-doppler signatures using wavelet analysis. In *IEEE High Performance Extreme Computing Conference (HPEC)*, September 2014.

[18] Swarun Kumar, Stephanie Gil, Dina Katabi, and Daniela Rus. Accurate indoor localization with zero start-up cost. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, pages 483–494. ACM, 2014.

[19] Kun-Chou Lee, Jhih-Sian Ou, and Ming-Chung Fang. Application of svd -reduction technique to pca based radar target recognition. *Progress In Electromagnetics Research*, 81:447–459, 2008.

[20] Yasamin Mostofi. Cooperative wireless-based obstacle/object mapping and see-through capabilities in robotic networks. *IEEE Transactions on Mobile Computing*, 12 (5):817–829, 2013.

[21] Rajalakshmi Nandakumar, Krishna Kant Chintalapudi, Venkat Padmanabhan, and Ramarathnam Venkatesan. Dhwani: secure peer-to-peer acoustic NFC. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 63–74. ACM, 2013.

[22] Jeffrey A Nanzer and Robert L Rogers. Bayesian classification of humans and vehicles using micro-doppler signals from a scanning-beam radar. *Microwave and Wireless Components Letters, IEEE*, 19(5):338–340, 2009.

[23] Byung-Kwon Park, Olga Boric-Lubecke, and Victor M Lubecke. Arctangent demodulation with DC offset compensation in quadrature doppler radar receiver systems. *IEEE Transactions on Microwave Theory and Techniques*, 55(5):1073–1079, 2007.

[24] Qifan Pu, Sidhant Gupta, Shyamnath Gollakota, and Shwetak Patel. Whole-home gesture recognition using wireless signals. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, pages 27–38. ACM, 2013.

[25] RG Raj, VC Chen, and R Lipps. Analysis of radar human gait signatures. *Signal Processing, IET*, 4(3):234–244, 2010.

[26] Graeme E Smith, Karl Woodbridge, and Chris J Baker. Radar micro-doppler signature classification using dynamic time warping. *IEEE Transactions on Aerospace and Electronic Systems*, 46(3):1078–1096, 2010.

[27] Thayananthan Thayaparan, Sumeet Abrol, Edwin Riseborough, LJ Stankovic, Denis Lamothe, and Grant Duff. Analysis of radar micro-doppler signatures from experimental helicopter and human data. *IET Radar, Sonar & Navigation*, 1(4):289–299, 2007.

[28] Guanhua Wang, Yongpan Zou, Zimu Zhou, Kaishun Wu, and Lionel M Ni. We can hear you with Wi-Fi! In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, pages 593–604. ACM, 2014.

[29] Wei Wang, Alex X Liu, Muhammad Shahzad, Kang Ling, and Sanglu Lu. Understanding and modeling of wifi signal based human activity recognition. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 65–76. ACM, 2015.

[30] Yan Wang, Jian Liu, Yingying Chen, Marco Gruteser, Jie Yang, and Hongbo Liu. E-eyes: device-free location-oriented activity identification using fine-grained WiFi signatures. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, pages 617–628. ACM, 2014.

[31] Yang Yang and Ness B Shroff. Scheduling in wireless networks with full-duplex cut-through transmission. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 2164–2172. IEEE, 2015.

[32] Yanzi Zhu, Yibo Zhu, Ben Y Zhao, and Haitao Zheng. Reusing 60ghz radios for mobile radar imaging. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 103–116. ACM, 2015.